Parallel Programming (Part I)

Yong-Ming Li

Intergraph

February, 2014

Brief history

- In 1981, IBM introduced its own PC that had a clock rate of 4.77 MHz.

Brief history

- In 1981, IBM introduced its own PC that had a clock rate of 4.77 MHz.
- In 1992, both Hewlett-Packard and Digital Equipment Corporation broke the difficult 100 MHz limit.

Brief history

- In 1981, IBM introduced its own PC that had a clock rate of 4.77 MHz.
- In 1992, both Hewlett-Packard and Digital Equipment Corporation broke the difficult 100 MHz limit.
- In March 6, 2000, AMD reached the 1GHz milestone.

Brief history

- In 1981, IBM introduced its own PC that had a clock rate of 4.77 MHz.
- In 1992, both Hewlett-Packard and Digital Equipment Corporation broke the difficult 100 MHz limit.
- In March 6, 2000, AMD reached the 1GHz milestone.
- In 2002, Intel Pentium 4 model was introduced as the first CPU with a clock rate of 3 GHz.

Brief history

- In 1981, IBM introduced its own PC that had a clock rate of 4.77 MHz.
- In 1992, both Hewlett-Packard and Digital Equipment Corporation broke the difficult 100 MHz limit.
- In March 6, 2000, AMD reached the 1GHz milestone.
- In 2002, Intel Pentium 4 model was introduced as the first CPU with a clock rate of 3 GHz.

Since then, chip makers have struggled to design CPUs that run much faster than about 3.5 GHz due to thermodynamic limits in current semiconductor process technologies.

Brief history

The major CPU vendors have shifted their attention away from ramping up clock speeds to adding parallelism support with multi-core processors that can handle different tasks simultaneously.

Brief history

The major CPU vendors have shifted their attention away from ramping up clock speeds to adding parallelism support with multi-core processors that can handle different tasks simultaneously.

The reason is that increasing performance through parallel processing can be far more energy-efficient than increasing microprocessor clock frequencies. In a world that is increasingly mobile and energy conscious, this has become essential.

Brief history

The major CPU vendors have shifted their attention away from ramping up clock speeds to adding parallelism support with multi-core processors that can handle different tasks simultaneously.

The reason is that increasing performance through parallel processing can be far more energy-efficient than increasing microprocessor clock frequencies. In a world that is increasingly mobile and energy conscious, this has become essential.

This seems like pretty good news. Yet, for most tasks, it is not. That is because the majority of software is traditionally designed to run on a single-core chip; in other words, it is designed to do only one thing at a time.

Brief history

As multi-core processors bring parallel computing to mainstream customers, the key challenge in computing today is transitioning the software industry to parallel programming.

Brief history

As multi-core processors bring parallel computing to mainstream customers, the key challenge in computing today is transitioning the software industry to parallel programming.

It requires broad collaborations across industry and academia to create families of technologies that work together to bring the power of parallel computing to future mainstream applications.

Brief history

As multi-core processors bring parallel computing to mainstream customers, the key challenge in computing today is transitioning the software industry to parallel programming.

It requires broad collaborations across industry and academia to create families of technologies that work together to bring the power of parallel computing to future mainstream applications.

The changes will affect the entire industry – from consumers to hardware manufacturers and from the entire software development infrastructure to application developers who rely upon it.

Brief history

As multi-core processors bring parallel computing to mainstream customers, the key challenge in computing today is transitioning the software industry to parallel programming.

It requires broad collaborations across industry and academia to create families of technologies that work together to bring the power of parallel computing to future mainstream applications.

The changes will affect the entire industry – from consumers to hardware manufacturers and from the entire software development infrastructure to application developers who rely upon it.

Parallel computing embodies multiple points of execution. It is more difficult to write a program that is divided into multiple concurrent tasks, due to the necessary synchronization and communication that need to take place between those tasks.

Models

There are multiple models for constructing parallel programs. At a very high level, there are two basic models:

Models

There are multiple models for constructing parallel programs. At a very high level, there are two basic models:

- *Multiple-program multiple-data* (MPMD). Each processor executes a different program ("multiple-program") with the different programs working in tandem to process different data ("multiple-data").

Models

There are multiple models for constructing parallel programs. At a very high level, there are two basic models:

- *Multiple-program multiple-data* (MPMD). Each processor executes a different program ("multiple-program") with the different programs working in tandem to process different data ("multiple-data").
- *Single-program multiple-data* (SPMD). The same program runs on each processor ("single-program") but each program instance processes different data ("multiple-data").

Models

There are multiple models for constructing parallel programs. At a very high level, there are two basic models:

- *Multiple-program multiple-data* (MPMD). Each processor executes a different program ("multiple-program") with the different programs working in tandem to process different data ("multiple-data").
- *Single-program multiple-data* (SPMD). The same program runs on each processor ("single-program") but each program instance processes different data ("multiple-data").

In this presentation, we are concerned only with the SPMD model.

Models

The two most common implementations of the SPMD model are:

Models

The two most common implementations of the SPMD model are:

- *Message Passing Interface* (MPI). The advantage of the MPI programming model is that the user has complete control over data distribution and process synchronization, permitting the optimization data locality and workflow distribution. The disadvantage is that existing sequential applications require a considerable amount of restructuring for a parallelization based on MPI.

Models

The two most common implementations of the SPMD model are:

- *Message Passing Interface* (MPI). The advantage of the MPI programming model is that the user has complete control over data distribution and process synchronization, permitting the optimization data locality and workflow distribution. The disadvantage is that existing sequential applications require a considerable amount of restructuring for a parallelization based on MPI.
- *Open Multi-Processing* (OpenMP). The advantage of OpenMP is that an existing code can be easily parallelized by placing OpenMP directives around time-consuming loops that do not contain data dependence, leaving the source code largely unchanged. Furthermore, parallelism directives can be added incrementally. The disadvantage is that it is not easy for the user to optimize workflow and memory access.

OpenMP overview

OpenMP helps C/C++ and FORTRAN developers create multi-threaded applications with minimal and manageable changes in their serial code. It is designed for *shared memory* systems like the ones we have in our desktop and laptop computers.

OpenMP overview

OpenMP helps C/C++ and FORTRAN developers create multi-threaded applications with minimal and manageable changes in their serial code. It is designed for *shared memory* systems like the ones we have in our desktop and laptop computers.

In computer hardware, shared memory refers to a large block of random access memory (RAM) that can be accessed by several different central processing units (CPUs) in a multiple-processor computer system.

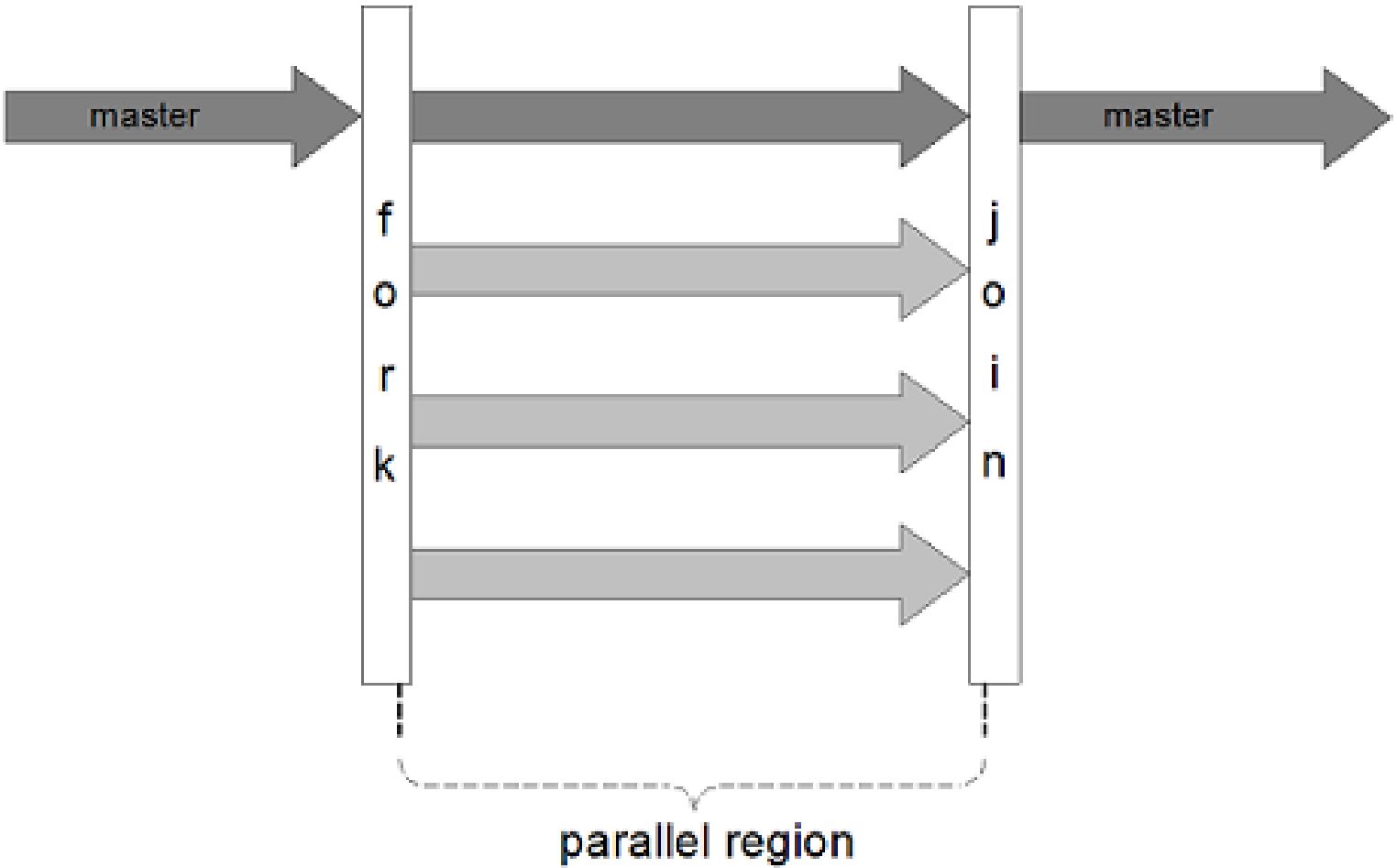
OpenMP overview

OpenMP helps C/C++ and FORTRAN developers create multi-threaded applications with minimal and manageable changes in their serial code. It is designed for *shared memory* systems like the ones we have in our desktop and laptop computers.

In computer hardware, shared memory refers to a large block of random access memory (RAM) that can be accessed by several different central processing units (CPUs) in a multiple-processor computer system.

OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization. It uses the so-called *fork-join* model of parallel execution as shown:

OpenMP overview



OpenMP overview

Let's write a short program to illustrate how OpenMP constructs a parallel regions and spawns a team of threads for parallel computing:

```
void main()
{
   printf("Multi-thread computing starts here:\n\n");
   #pragma omp parallel
   {
      printf("Hello from thread %d\n", omp_get_thread_num());
   }
   printf("\nMulti-thread computing ends here.\n");
}
```

If your computer has at least a dual-core processor, you should see more than one "Hello" line being printed at the command prompt.

OpenMP overview

My laptop has eight threads (Core i7) and, hence, prints:

```
Multi-thread computing starts here:

Hello from thread 0
Hello from thread 7
Hello from thread 6
Hello from thread 5
Hello from thread 3
Hello from thread 4
Hello from thread 2
Hello from thread 1

Multi-thread computing ends here.
```

OpenMP overview

This simple example was created to show how OpenMP's fork-join model behaves. In practice, we should not construct a parallel region to perform redundant work as the above program does.

OpenMP overview

This simple example was created to show how OpenMP's fork-join model behaves. In practice, we should not construct a parallel region to perform redundant work as the above program does.

The parallelism is usually constructed by using more than one `pragma` directive along with optional directive clauses to *share* (not to duplicate) the workload among available threads. The general syntax is:

```
#pragma omp <directive> [clause ...]
```

OpenMP overview

This simple example was created to show how OpenMP's fork-join model behaves. In practice, we should not construct a parallel region to perform redundant work as the above program does.

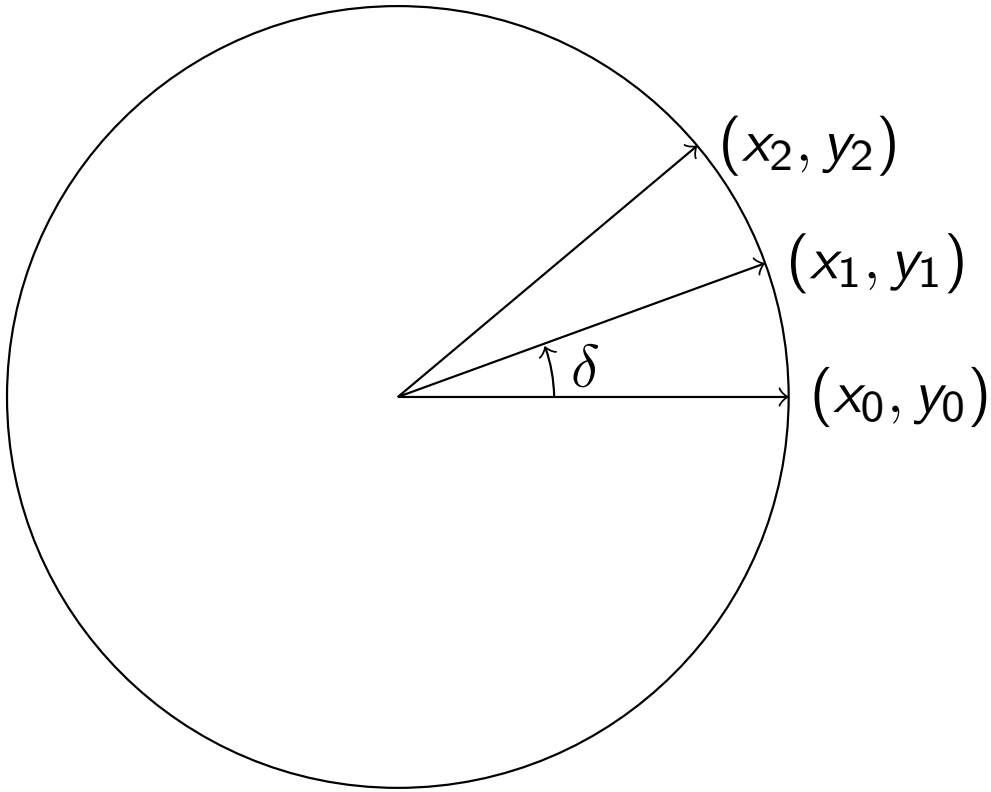
The parallelism is usually constructed by using more than one `pragma` directive along with optional directive clauses to *share* (not to duplicate) the workload among available threads. The general syntax is:

```
#pragma omp <directive> [clause ...]
```

Commonly-used directives are: **parallel**, **for**, **section**, **ordered**, **critical**, etc. The OpenMP clauses are optional modifiers of the directives and affect the behavior of the directives. The data sharing attribute clauses include **private**, **firstprivate**, **lastprivate**, and **reduction**. They are mainly used to manage data environment to avoid data race conditions.

OpenMP overview

To demonstrate how OpenMP constructs a parallel region to share the workload, we start by considering the circle stroking algorithm.



OpenMP overview

If we do not care about the performance, the circle stroking method may be implemented as follows:

```
void apiStrokeCircle(GPosition2d &center, double radius, double tol,
                     std::vector<GPosition2d>& pStrkPts, apiError &rc)
{
   int          i, nStrkPts;
   double       delta;

   // Determine the number of stroking points
   rc = api_OK;
   delta = 2.0 * acos(1.0 - tol / radius);
   nStrkPts = 2 + (int)(2.0 * GPI / delta);
   delta = 2.0 * GPI / (nStrkPts - 1);
   pStrkPts.resize(nStrkPts);

   // Loop to compute stroking points
   for (i=0; i<nStrkPts; i++)
   {
      pStrkPts[i].x = center.x + radius * cos(i * delta);
      pStrkPts[i].y = center.y + radius * sin(i * delta);
   }
}
```

OpenMP overview

If we do not care about the performance, the circle stroking method may be implemented as follows:

```
void apiStrokeCircle(GPosition2d &center, double radius, double tol,
                     std::vector<GPosition2d>& pStrkPts, apiError &rc)
{
   int          i, nStrkPts;
   double       delta;

   // Determine the number of stroking points
   rc = api_OK;
   delta = 2.0 * acos(1.0 - tol / radius);
   nStrkPts = 2 + (int)(2.0 * GPI / delta);
   delta = 2.0 * GPI / (nStrkPts - 1);
   pStrkPts.resize(nStrkPts);

   // Loop to compute stroking points
   for (i=0; i<nStrkPts; i++)
   {
      pStrkPts[i].x = center.x + radius * cos(i * delta);
      pStrkPts[i].y = center.y + radius * sin(i * delta);
   }
}
```

The computation of the ith stroking point is independent of the $(i-1)$th stroking point. Therefore, stroking points do not have to be computed sequentially.

OpenMP overview

We thus parallelize the method as follows:

```
void apiStrokeCircle(GPosition2d &center, double radius, double tol,
                     std::vector<GPosition2d>& pStrkPts, apiError &rc)
{
   int          i, nStrkPts;
   double       delta;

   // Determine the number of stroking points
   rc = api_OK;
   delta = 2.0 * acos(1.0 - tol / radius);
   nStrkPts = 2 + (int)(2.0 * GPI / delta);
   delta = 2.0 * GPI / (nStrkPts - 1);
   pStrkPts.resize(nStrkPts);

   // Loop to compute stroking points
   #pragma omp parallel
   {
      #pragma omp for
      for (i=0; i<nStrkPts; i++)
      {
          pStrkPts[i].x = center.x + radius * cos(i * delta);
          pStrkPts[i].y = center.y + radius * sin(i * delta);
      }
   }
}
```

OpenMP overview

We thus parallelize the method as follows:

```
void apiStrokeCircle(GPosition2d &center, double radius, double tol,
                     std::vector<GPosition2d>& pStrkPts, apiError &rc)
{
   int          i, nStrkPts;
   double       delta;

   // Determine the number of stroking points
   rc = api_OK;
   delta = 2.0 * acos(1.0 - tol / radius);
   nStrkPts = 2 + (int)(2.0 * GPI / delta);
   delta = 2.0 * GPI / (nStrkPts - 1);
   pStrkPts.resize(nStrkPts);

   // Loop to compute stroking points
   #pragma omp parallel
   {
      #pragma omp for
      for (i=0; i<nStrkPts; i++)
      {
          pStrkPts[i].x = center.x + radius * cos(i * delta);
          pStrkPts[i].y = center.y + radius * sin(i * delta);
      }
   }
}
```

The second work-sharing directive tells OpenMP that the `for` loop should have its iterations divided among the thread team.

OpenMP overview

My computer has four cores and eight hyper threads. Compiling and running the above parallel implementation, we would expect it to significantly outperform the one without parallelism.

OpenMP overview

My computer has four cores and eight hyper threads. Compiling and running the above parallel implementation, we would expect it to significantly outperform the one without parallelism.

In reality, it is only 2 times faster. This is because multi-threaded executables often incur longer startup times. In some cases, they can actually run much slower than if compiled by a single thread!

OpenMP overview

My computer has four cores and eight hyper threads. Compiling and running the above parallel implementation, we would expect it to significantly outperform the one without parallelism.

In reality, it is only 2 times faster. This is because multi-threaded executables often incur longer startup times. In some cases, they can actually run much slower than if compiled by a single thread!

Based on Microsoft's article, multi-threading computation may not be appropriate if the `for` loop block runs less than 15 millisecond. Therefore, it is important to understand and evaluate the workload before adding parallelism.

Race condition

In writing parallel programs, understanding what data is shared and what is private becomes very important not only for performance, but also for correct operation.

Race condition

In writing parallel programs, understanding what data is shared and what is private becomes very important not only for performance, but also for correct operation.

Shared variables are shared by all threads in the thread team. Thus, a change of any shared variable in one thread becomes visible to all the threads in the parallel region.

Race condition

In writing parallel programs, understanding what data is shared and what is private becomes very important not only for performance, but also for correct operation.

Shared variables are shared by all threads in the thread team. Thus, a change of any shared variable in one thread becomes visible to all the threads in the parallel region.

Private variables, on the other hand, have private copies made for each thread in the thread team, so changes made in one thread are not visible to the other threads.

Race condition

To understand how data scope may affect parallel programming, we consider the evaluation of the trigonometric function $\sin x$.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)!} x^{2i+1}.$$

Race condition

To understand how data scope may affect parallel programming, we consider the evaluation of the trigonometric function $\sin x$.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)!} x^{2i+1}.$$

It is seen that the ith term is equal to the $(i-1)$th term multiplied by

$$x^2/i(i-1)$$

with the sign negated.

Race condition

To understand how data scope may affect parallel programming, we consider the evaluation of the trigonometric function $\sin x$.

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)!} x^{2i+1}.$$

It is seen that the ith term is equal to the $(i-1)$th term multiplied by

$$x^2/i(i-1)$$

with the sign negated. So, the evaluation of $\sin x$ is:

1. Assign x to the intermediate variable `term` and set $f = 0$ and $i = 3$.
2. Perform `f += term`.
3. Multiply `term` by $x^2/(i(i-1))$ and negate its sign.
4. Increment i by 2 and go to step 2 to repeat the process until convergence is reached.

Race condition

Evaluation of $\sin x$ at n evenly-spaced points is implemented as follows:

```
void apiEvalSine(int numPts, double *pValues)
{
   int         i, n;
   double      x, xx, f, term, delta;

   delta = 2.0 * GPI / (numPts - 1);
   for (n=0; n<numPts; n++)
   {
      x = n * delta;
      i = 3;
      f = 0.0;
      term = x;
      xx = x * x;
      while (true)
      {
         f += term;
         if (fabs(term) < DBL_EPSILON)
         {
            pValues[n] = f;
            break;
         }
         term = -term * xx / (i * (i - 1));
         i += 2;
      }
   }
}
```

Race condition

Construct the parallel region to share the iterations:

```
void apiEvalSine(int numPts, double *pValues)
{
   int         i, n;
   double      x, xx, f, term, delta;

   delta = 2.0 * GPI / (numPts - 1);
   #pragma omp parallel
   {
      #pragma omp for
      for (n=0; n<numPts; n++)
      {
         x = n * delta;
         i = 3;
         f = 0.0;
         term = x;
         xx = x * x;
         while (true)
         {
            f += term;
            if (fabs(term) < DBL_EPSILON)
            {
               pValues[n] = f;
               break;
            }
            term = -term * xx / (i * (i - 1));
            i += 2;
         }
      }
   }
}
```

Race condition

Compiling and running the above implementation at the command line, we will see

```
D:\ArtProgram\test>mntest EvalSine 1 1

 Processing data file EvalSine.d1 ...

CPU time = 0.000000 seconds for 1 iteration.
         *** Results differ from previous ones.
```

Race condition

Compiling and running the above implementation at the command line, we will see

```
D:\ArtProgram\test>mntest EvalSine 1 1

 Processing data file EvalSine.d1 ...

CPU time = 0.000000 seconds for 1 iteration.
         *** Results differ from previous ones.
```

How can this happen by simply declaring parallelism in the working code?

Race condition

Compiling and running the above implementation at the command line, we will see

```
D:\ArtProgram\test>mntest EvalSine 1 1

 Processing data file EvalSine.d1 ...

CPU time = 0.000000 seconds for 1 iteration.
         *** Results differ from previous ones.
```

How can this happen by simply declaring parallelism in the working code?

It happens because every thread is racing to *modify the shared* variables: `i, x, xx, f, term`.

Race condition

One possible fix is to declare shared variables as private:

```
void apiEvalSine(int numPts, double *pValues)
{
   int      i, n;
   double   x, xx, f, term, delta;

   delta = 2.0 * GPI / (numPts - 1);
   #pragma omp parallel
   {
      #pragma omp for private (i, x, xx, f, term)
      for (n=0; n<numPts; n++)
      {
         x = n * delta;
         i = 3;
         f = 0.0;
         term = x;
         xx = x * x;
         while (true)
         {
            f += term;
            if (fabs(term) < DBL_EPSILON)
            {
               pValues[n] = f;
               break;
            }
            term = -term * xx / (i * (i - 1));
            i += 2;
         }
      }
   }
}
```

Race condition

My recommendation is to make a sub-function out of the block so that the code is modularized and the `for` loop is simplified as shown below:

```
void apiEvalSine(int numPts, double *pValues)
{
   int          n;
   double       delta;

   delta = 2.0 * GPI / (numPts - 1);
   #pragma omp parallel
   {
      #pragma omp for
      for (n=0; n<numPts; n++)
      {
         evalOneSine(n * delta, pValues[n]);
      }
   }
}
```

Race condition

Making variables private does not solve all race and synchronization problems. A common example is to have a summation in the `for` loop:

$$x_c = \frac{\sum_{i=1}^{n} x_i}{n}, \quad y_c = \frac{\sum_{i=1}^{n} y_i}{n}, \quad z_c = \frac{\sum_{i=1}^{n} z_i}{n}.$$

Race condition

Making variables private does not solve all race and synchronization problems. A common example is to have a summation in the `for` loop:

$$x_c = \frac{\sum_{i=1}^{n} x_i}{n}, \quad y_c = \frac{\sum_{i=1}^{n} y_i}{n}, \quad z_c = \frac{\sum_{i=1}^{n} z_i}{n}.$$

Such computations can be implemented as

```
void apiCenterOfPoints(std::vector<GPosition> &points, GPosition &center)
{
   int      i, n;
   double   xSum, ySum, zSum;

   n = (int)points.size();
   xSum = ySum = zSum = 0.0;

   for (i=0; i<n; i++)
   {
      xSum += points[i].x;
      ySum += points[i].y;
      zSum += points[i].z;
   }

   center.x = xSum / n;
   center.y = ySum / n;
   center.z = zSum / n;
}
```

Race condition

When the `for` loop is parallelized, all threads will race to update the three scalar variables and, hence, may produce an unpredictable result.

Race condition

When the `for` loop is parallelized, all threads will race to update the three scalar variables and, hence, may produce an unpredictable result.

If `xSum`, `ySum`. and `zSum` are declared to be private, each thread will get their own copies of the variables. Accordingly, the race condition seems to be solved.

Race condition

When the `for` loop is parallelized, all threads will race to update the three scalar variables and, hence, may produce an unpredictable result.

If `xSum`, `ySum`. and `zSum` are declared to be private, each thread will get their own copies of the variables. Accordingly, the race condition seems to be solved.

However, it is not for two reasons:

Race condition

When the `for` loop is parallelized, all threads will race to update the three scalar variables and, hence, may produce an unpredictable result.

If `xSum`, `ySum`. and `zSum` are declared to be private, each thread will get their own copies of the variables. Accordingly, the race condition seems to be solved.

However, it is not for two reasons:

- Local copies are destroyed when threads complete their assigned work.

Race condition

When the `for` loop is parallelized, all threads will race to update the three scalar variables and, hence, may produce an unpredictable result.

If `xSum`, `ySum`. and `zSum` are declared to be private, each thread will get their own copies of the variables. Accordingly, the race condition seems to be solved.

However, it is not for two reasons:

- Local copies are destroyed when threads complete their assigned work.
- Local copies hold partial summation results that need to be combined to generate a correct result.

Race condition

OpenMP solves this kind of problems by providing the following **reduction** clause:

```
void apiCenterOfPoints(std::vector<GPosition> &points,
                                    GPosition &center)
{
   int      i, n;
   double   xSum, ySum, zSum;

   n = (int)points.size();
   xSum = ySum = zSum = 0.0;

   #pragma omp parallel
   {
      #pragma omp for reduction(+ : xSum,ySum,zSum)
      for (i=0; i<n; i++)
      {
         xSum += points[i].x;
         ySum += points[i].y;
         zSum += points[i].z;
      }
   }

   center.x = xSum / n;
   center.y = ySum / n;
   center.z = zSum / n;
}
```

Race condition

It is stated in many OpenMP tutorials and documents that all variables in a parallel region are shared by default, with some exceptions such as `for` loop index and locally-defined variables.

Race condition

It is stated in many OpenMP tutorials and documents that all variables in a parallel region are shared by default, with some exceptions such as `for` loop index and locally-defined variables.

What about the indices of nested `for` loops?

Race condition

It is stated in many OpenMP tutorials and documents that all variables in a parallel region are shared by default, with some exceptions such as `for` loop index and locally-defined variables.

What about the indices of nested `for` loops?

In FORTRAN, each thread gets a private copy of the loop index for any loops nested inside the main loop.

Race condition

It is stated in many OpenMP tutorials and documents that all variables in a parallel region are shared by default, with some exceptions such as `for` loop index and locally-defined variables.

What about the indices of nested `for` loops?

In FORTRAN, each thread gets a private copy of the loop index for any loops nested inside the main loop.

In C/C++, nested loop indices are not automatically "privatized." By default, only the outer `for` loop index is private, and all sequential loop indices are shared.

Race condition

For this reason, we parallelize the product of two matrices as follows:

```
void apiAtAMatrixV1(int n, int m, double **pA, double **pAtA)
{
   int      i, j, k;

   #pragma omp parallel
   {
      #pragma omp for private (j, k)
      for (i=0; i<m; i++)
      {
         for (j=0; j<m; j++)
         {
            pAtA[i][j] = 0.0;
            for (k=0; k<n; k++)
            {
               pAtA[i][j] += pA[k][i] * pA[k][j];
            }
         }
      }
   }
}
```

Race condition

For this reason, we parallelize the product of two matrices as follows:

```
void apiAtAMatrixV1(int n, int m, double **pA, double **pAtA)
{
   int      i, j, k;

   #pragma omp parallel
   {
      #pragma omp for private (j, k)
      for (i=0; i<m; i++)
      {
         for (j=0; j<m; j++)
         {
            pAtA[i][j] = 0.0;
            for (k=0; k<n; k++)
            {
               pAtA[i][j] += pA[k][i] * pA[k][j];
            }
         }
      }
   }
}
```

It is noted that only the iterative workload of the outer loop is divided among threads. The performance may be further improved by using the **collapse** clause available only in OpenMP 3.0 and newer versions.

Load balance

Load balancing (i.e., the division of work among threads) is one of the most important attributes for parallel application performance.

Load balance

Load balancing (i.e., the division of work among threads) is one of the most important attributes for parallel application performance.

OpenMP, by default, assumes that all loop iterations consume the same amount of time. This assumption leads OpenMP to distribute the iterations of the loop among the threads in roughly equal amounts. If some threads complete their tasks, they will pause until the last thread finishes before leaving the parallel region due to the implicit barrier synchronization at the end of the parallel region. This obviously degrades the overall performance.

Load balance

Load balancing (i.e., the division of work among threads) is one of the most important attributes for parallel application performance.

OpenMP, by default, assumes that all loop iterations consume the same amount of time. This assumption leads OpenMP to distribute the iterations of the loop among the threads in roughly equal amounts. If some threads complete their tasks, they will pause until the last thread finishes before leaving the parallel region due to the implicit barrier synchronization at the end of the parallel region. This obviously degrades the overall performance.

To ensures the processors are busy most of the time, OpenMP provides a collections of clauses to optimize the performance.

Load balance

Referring to `apiEvalSine` algorithm, the Taylor series of $\sin x$ converges faster when x is small. If the `for` loop are evenly divided among available threads, some threads will complete sooner than others.

Load balance

Referring to `apiEvalSine` algorithm, the Taylor series of $\sin x$ converges faster when x is small. If the `for` loop are evenly divided among available threads, some threads will complete sooner than others.

We can add the clause **nowait** to suppress implicit barrier synchronization and, hence, improve the performance.

```
void apiEvalSine(int numPts, double *pValues)
{
   int          i;
   double       delta = 2.0 * GPI / (numPts - 1);
   #pragma omp parallel
   {
      #pragma omp for nowait
      for (i=0; i<numPts; i++)
      {
         evalOneSine(i * delta, pValues[i]);
      }
   }
}
```

Load balance

OpenMP offers also the **schedule** clause with a set of predefined iteration scheduling strategies to fine-tune the performance. The syntax is

```
#pragma omp parallel for schedule(type [, chunk size])
```

There are four different loop scheduling types:

Load balance

OpenMP offers also the **schedule** clause with a set of predefined iteration scheduling strategies to fine-tune the performance. The syntax is

```
#pragma omp parallel for schedule(type [, chunk size])
```

There are four different loop scheduling types:

static Divides the loop into equal-sized chunks or as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, the chunk size is loop / number of threads.

Load balance

OpenMP offers also the **schedule** clause with a set of predefined iteration scheduling strategies to fine-tune the performance. The syntax is

```
#pragma omp parallel for schedule(type [, chunk size])
```

There are four different loop scheduling types:

static Divides the loop into equal-sized chunks or as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, the chunk size is loop / number of threads.

dynamic Uses an internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retries the next block of loop iterations from the top of the work queue. By default, chunk size is 1. Be careful when using this scheduling type because of the extra overhead required.

Load balance

guided Similar to dynamic scheduling, but the chunk size starts off large and shrinks in an effort to reduce the amount of time threads have to go to the work-queue to get more work. The optional chunk parameter specifies the minimum chunk to use. By default, the chunk size is approximately the loop count / number of threads.

Load balance

guided Similar to dynamic scheduling, but the chunk size starts off large and shrinks in an effort to reduce the amount of time threads have to go to the work-queue to get more work. The optional chunk parameter specifies the minimum chunk to use. By default, the chunk size is approximately the loop count / number of threads.

runtime The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

Load balance

guided Similar to dynamic scheduling, but the chunk size starts off large and shrinks in an effort to reduce the amount of time threads have to go to the work-queue to get more work. The optional chunk parameter specifies the minimum chunk to use. By default, the chunk size is approximately the loop count / number of threads.

runtime The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

It is not an easy task to choose the appropriate assignment of loop iterations to threads.

Load balance

guided Similar to dynamic scheduling, but the chunk size starts off large and shrinks in an effort to reduce the amount of time threads have to go to the work-queue to get more work. The optional chunk parameter specifies the minimum chunk to use. By default, the chunk size is approximately the loop count / number of threads.

runtime The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

It is not an easy task to choose the appropriate assignment of loop iterations to threads.

In practice, it is recommended to perform various timing tests and choose the one that has the best overall performance.

Data synchronization, non-loop parallelism, and explicit task parallelism will be discussed in the next presentation. Two application examples used in S3D will also be reviewed.

Data synchronization, non-loop parallelism, and explicit task parallelism will be discussed in the next presentation. Two application examples used in S3D will also be reviewed.

The presentation is extracted from Chapter 11 of my book:

A Practical Guide to Developing Computational Software.

It is sold at Amazon.com and Barnes & Noble.