How to parallelize the code of your choice

By David Livshin, published on August 29, 2021

Copyright © 2021 David Livshin. All rights reserved.
david.livshin@dalsoft.com

In this article we will show how to apply the technology developed in ***dpl*** to parallelize the code of your choice.

First we will give a brief overview of the ***dlp*** library and then explain how, using technology developed during implementation of ***dpl***, to parallelize benchmarks, DSP filters and routines from computational (quantitative) finance.

Note that many routines mentioned here are not part of the dpl library, please contact us at support@dalsoft.com to discuss the parallel implementation of the serial code of your choice.

Brief overview of dpl

dpl - Dalsoft's Parallel Library - is a software library that allows parallel execution of a number of sequential functions (**stencils**) - see this for full description. ***dpl*** is designed to be used on multi core processors with the operating environment that supports OpenMP.

dpl provides the following sets of routines - **a**, **b**, **c**, **x**... being input/output vectors/matrices of an appropriate size (**n**, or **nxn**):

Routines for parallel implementation of basic stencils

dpl_stencil_1p

x[i] = a[i]*x[i-stride] + b[i] for **i** = **stride**...**n-1**

dpl_stencil_o_1p

x[i] = c[i]*x[i] + a[i]*x[i-stride] + b[i] for **i** = **stride**...**n-1**

dpl_stencil_div_1p

x[i] = a[i]/x[i-stride] + b[i] for **i** = **stride**...**n-1**

dpl_stencil_div_o_1p

x[i] = c[i]*x[i] + a[i]/x[i-stride] + b[i] for **i** = **stride**...**n-1**

dpl_stencil_acc

acc = a[i]*acc + b[i] for **i** = **stride**...**n-1**

dpl_stencil_div_acc

acc = a[i]/acc + b[i] for **i** = **stride**...**n-1**

dpl_stencil_2p

x[i] = a[i]*x[i-1] * c[i]*x[i-2] + b[i] for **i** = **2**...**n-1**

dpl_stencil

x[i] = b[i] + $\sum_{j=0}^{i-1}$ **a[i][j]*x[j]** for **i** = **0**...**n-1**.

Routines for parallel implementation of conditional/stochastic functions

dpl_stencil_min

x[i] = min(a[i]*x[i-1], b[i]) for **i** = **1**...**n-1**

dpl_stencil_max

x[i] = max(a[i]*x[i-1], b[i]) for **i** = **1**...**n-1**

dpl_max_st

computes the stochastic ordering of the input vectors **a** and **b** of the size **n** calculating the output vector **x** of the size **n**: **x** = $\mathbf{max}_{\mathbf{st}}$**(a, b)**
as the solution of the following system of equations:
for every **i** = **0**...**n-1**: $\sum_{j=i}^{n-1}$ **x[j] = max(** $\sum_{j=i}^{n-1}$ **a[j],** $\sum_{j=i}^{n-1}$ **b[j])**

dpl_stochastic_max

calculates the output matrix **x** of the size **n**x**n** to be the stochastic matrix of the input matrix **b** of the size **n**x**n** as the following stencil:

(with m[j,*] representing j's row of the matrix m, $\mathbf{max}_{\mathbf{st}}$ being defined in the previous paragraph**)**

x[0,*] = b[0,*]
x[i,*] = $\mathbf{max}_{\mathbf{st}}$**(x[i-1,*], b[i,*])** for **i** = **1**...**n-1**

Routine for parallel implementation of the Gauss-Seidel method to solve systems of linear equations

dpl_solve_gs

solves a square system of **n** linear equations **ax = b** using Gauss–Seidel method.
Implements interactive process with the next iteration **k+1** $\mathbf{x}^{\mathbf{k+1}}$ being calculated by applying Gauss-Seidel method to the value of the initial/previous iterations $\mathbf{x}^{\mathbf{k}}$. After that the norm

$\varepsilon_{k+1} = | x^{k+1} - x^k |_\infty = \max(|x^{k+1}[i] - x^k[i]|)$ for $i = 0,...,n-1$

is calculated and process continues till ε_{k+1} becomes less than the specified **tolerance**.

Routine for parallel implementation of the general 2-D 5-points stencil

dpl_p1p0_p0p1_p0p0_p0n1_n1p0

for an elemnt (**r**,**c**) of a 2-dimensional matrix define elements as following:

[pn]#1[pn]#2

meaning

p - previous or the same

n - next

- number of elements from (**r**,**c**) according to [**pn**]

for example **p1n2** is the element (**r**-1,**c**+2)

in the following grid, that defines 2-D 5-points stencil

	c-1	**c**	**c**+1
r-1		**r**-1,**c**	
r	**r**,**c**-1	**r**,**c**	**r**,**c**+1
r+1		**r**+1,**c**	

the elements will be

r-1,**c**	**p1p0**
r,**c**-1	**p0p1**
r,**c**	**p0p0**
r,**c**+1	**p0n1**
r+1,**c**	**n1p0**

DESCRIPTION

computes

```
for ( r = 1 ; r < rows - 1; r++ )
 {
  for ( c = 1 ; c < cols - 1 ; c++ )
   {
     x[r][c] = x[r-1][c]*ap1p0[r][c] +
               x[r][c-1]*ap0p1[r][c] +
               x[r][c]*ap0p0[r][c] +
               x[r][c+1]*ap0n1[r][c] +
               x[r+1][c]*an1p0[r][c] +
               b[r][c];
   }
```

```
}
```

Auxiliary routines

dpl_aux_strerror

returns a pointer to a string that describes the error code generated by the last call to a parallel routine from the ***dpl*** library;

applying the technology developed in dpl

The following shows the results of applying routines from the ***dpl*** in order to optimize benchmarks and/or applications. Note that many routines shown here do not appear as stencils but nevertheless benefit from the use of stencil library,.

adi from the Polybench

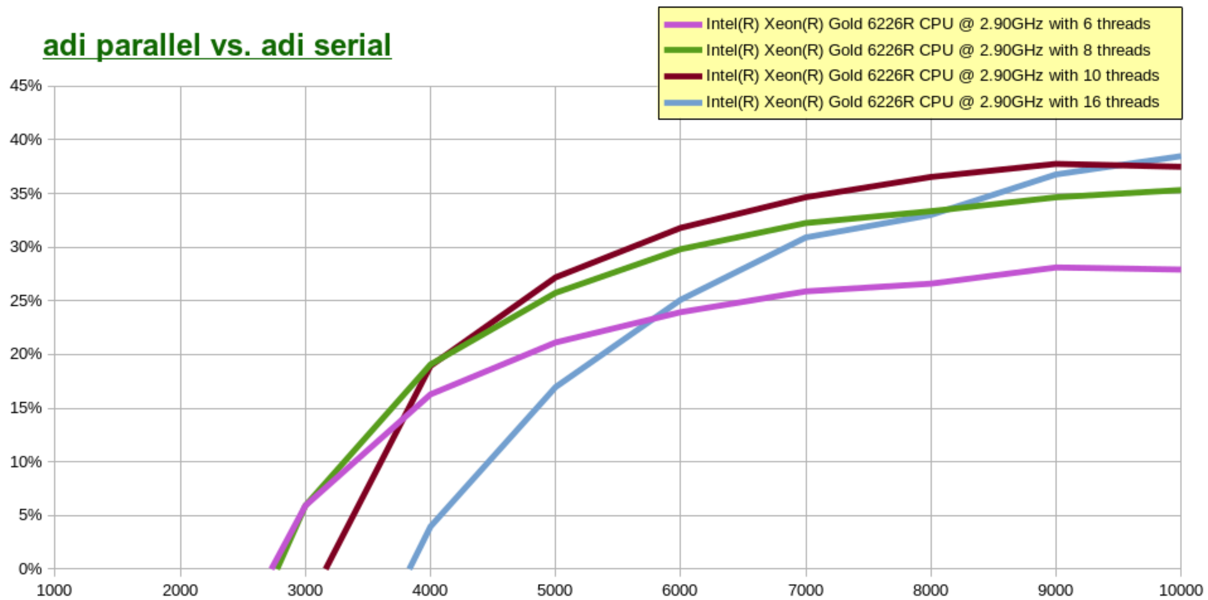
Here we study an **adi** benchmark from the PolyBench benchmark suite - see this for information about this benchmark suite: disclaimer, copyright and how to download.

We used the following code segment from the **adi** benchmark of the PolyBench benchmark suite:

```
for ( i1 = 0; i1 < _PB_N; i1++ )
 for ( i2 = 1; i2 < _PB_N; i2++ )
  {
   X[i1][i2] = X[i1][i2] - X[i1][i2-1] * A[i1][i2] / B[i1][i2-1];
   B[i1][i2] = B[i1][i2] - A[i1][i2] * A[i1][i2] / B[i1][i2-1];
  }
```

The code was slightly modified to allow use of the routines from the ***dpl*** library.
The following graph presents results of the performance comparison between modified and the original codes.

- The X axis lists different sizes for the problem we attempted to study (the value of **_PB_N**)
- The Y axis lists percent of improvement achieved by the modified code that uses ***dpl*** library over the original serial one
- The legend specifies on what CPU the plotted data was generated



latnrm - DSP filter

We downloaded the code for the **ORDER** order Normalized Lattice filter processing **NPOINTS** points (**latnrm**) - we will refer to this code as ***serial version*** of the function:

```
int i, j;
double left,right,top;
double bottom=0;
double sum;
for (i = 0; i < NPOINTS; i++)
 {
  top = InpData[i];
  for (j = 1; j < ORDER; j++)
   {
    left = top;
    right = InternalState[j];
    InternalState[j] = bottom;
    top = Coefficients[j-1] * left - Coefficients[j] * right;
    bottom = Coefficients[j-1] * right + Coefficients[j] * left;
   }
  InternalState[ORDER] = bottom;
  InternalState[ORDER+1] = top;
  sum = 0.0;
  for (j = 0; j < ORDER; j++)
   {
```

```
    sum += InternalState[j] * Coefficients[j+ORDER];
   }
  OutData[i] = sum;
 }
```

As it appears above, it is difficult to see any code patterns (stencils)
that might allow to apply our technology.
However the code may be rewritten as (only loops are shown)

```
for ( Data_indx = 0; Data_indx < NPOINTS; Data_indx++ )
 {
  left = InpData[Data_indx];
  for ( j = 1; j < ORDER; j++ )
   {
    InternalState[j+1] = Coefficients[j-1] * InternalState[j] +
      Coefficients[j] * left;
    left = Coefficients[j-1] * left - Coefficients[j] *
       InternalState[j];
   }

  InternalStateS[Order+1] = left;

  sum = 0.0;
  for ( j = 0; j < ORDER; j++ )
   {
    sum += InternalState[j] * Coefficients[j+ORDER];
   }
  OutData[Data_indx] = sum;
 }
```

which makes it possible for a number of the available techniques to be applied. Note that in this case none of the routines from the ***dpl*** library may be used directly. However the technology developed for the implementation of this library may be used here.

We parallelized the one-point stencil

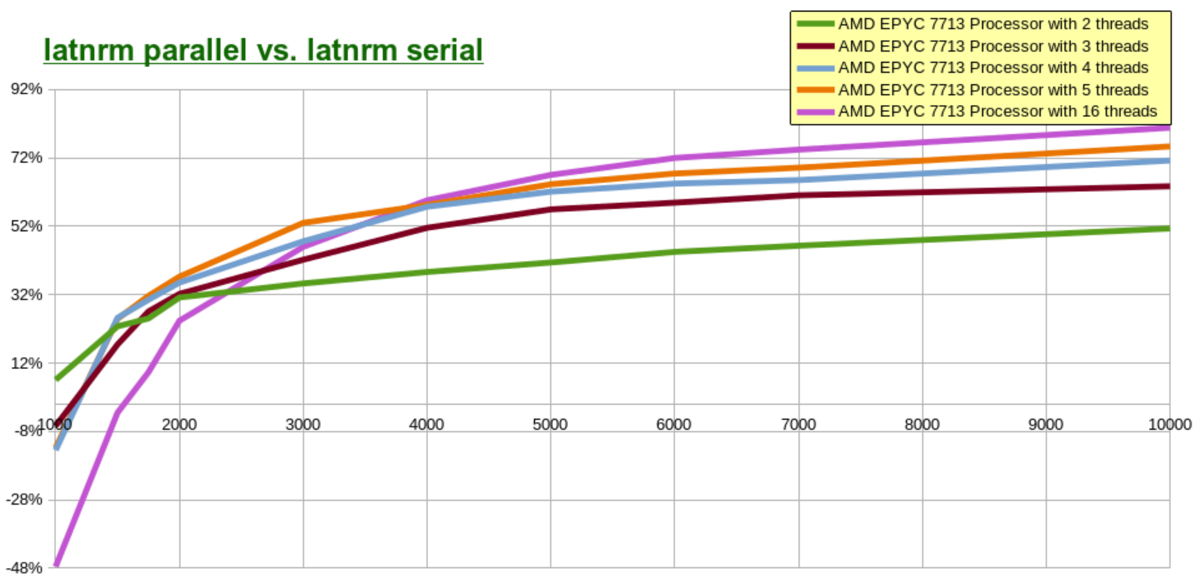
left = Coefficients[j-1] * left - Coefficients[j] * InternalState[j]

overlapping the generated parallel code with the rest of the routine.
This led to the generation of the parallel code for the **latnrm** DSP filter listed above.

The evaluation of the created parallel code was done under Linux operating system running on the **AMD EPYC 7713** Processor with 16 threads.

The following graph presents improvement of the created parallel implementation of the function **latnrm** over it serial version.

- The X axis lists different sizes for the problem we attempted to study(the value of **NPOINTS**; **ORDER** was always set to **NPOINTS**>>1; the input arrays consist of randomly generated double precision values; generation of random values was done using Dalsoft's Random Testing (***drt***) package - see this for more information).
- The Y axis lists percent of improvement (or otherwise) achieved by the created parallel implementation of the function **latnrm** over it serial version;
- The legend specifies on what CPU the plotted data was generated



Note that change between execution time of the created parallel implementation of the function **latnrm** and it serial version depends on the size of the problem (the value of **NPOINTS**) as well as the number of threads utilized. Using more threads doesn't always result in faster execution time:

using 16 threads leads for fastest execution time only if the value of **NPOINTS** is **4000** or greater; for smaller values of **NPOINTS** (e.g. **1000**), the best results achieved with the smaller number of threads utilized (e.g. 2 threads for **NPOINTS** being **1000**).

See this for similar study.

Computational (quantitative) finance

Here we will analyze some of the code used in computational (quantitative) finance and show how the technology, developed in ***dpl***, may be applied to improve the performance of these applications through parallelization.

The following contains parallel implementation for the

- Leisen-Reimer binomial tree: European Call and American Call
- Crank-Nicolson finite difference method for option pricing: European Put
- Heston stochastic volatility model: European Call

Please note that routines shown here are not part of the dpl package. Please Contact us to discuss the parallel implementation of a serial code of your choice.

Leisen-Reimer binomial tree

We use the following C++ code, developed by Dr. Fabrice Douglas Rouah (see this
We use the following C++ code, developed by Dr. Fabrice Douglas Rouah (visit this for general reference, see here for the code itself), as the base for our study.

The following optimizes European and American Call options. The corresponding Put options can be handled similarly.

Leisen-Reimer binomial tree, European Call

The European Call option is implemented by the following code portion of the original code

```
for ( j = n - 1; j >= 0; j-- )
 {
 for ( i = 0; i <= j; i++ )
  {
  EC[i][j] = exp(-r*dt)*(p*EC[i][j + 1] + (1 - p)*EC[i + 1][j + 1]);
  }
 }
```

with **n** being number of time steps utilized.
We will refer to the executable derived from this code as *serial* implementation.

The internal loop of the above code is parallel and may be easily parallelized using OpenMP - we will refer to the executable derived from that parallel version of code as *quick_parallel* implementation.

However there is a way to create a better parallel code.
Perform Loop interchange of the above code getting the loop

```
for ( i = n - 1; i >= 0; i-- )
 {
 for ( j = n - 1; j <= i; j-- )
```

```
  {
   EC[i][j] = exp(-r*dt)*(p*EC[i][j + 1] + (1 - p)*EC[i + 1][j + 1]);
  }
 }
```

Now the internal loop is not parallel but it is a stencil (reminiscent to the one-point stencil **dpl_stencil_1p**) and the technology, developed in ***dpl***, may be applied to parallelize and improve this code.
The major sources of optimization of the loop after interchange are:

- cache friendly memory access
- one parallel invocation for all the code
- the ability to create loop pipeline for the encompassing `for ( i = n - 1; i >= 0; i-- )` loop

Combined with the careful coding we obtain an executable which is refereed as *parallel* implementation.

The following timing improvement was observed on the **Intel® Core(TM) i5-9400** Processor with 6 threads. Each column represents the number of time steps utilized. Each row represents improvement of the code of the first specified executable type over the code of the second executable type.
The following table summarizes the above data and provides improvements of the optimized/parallelized code over original sequential code.

	1001	*1501*	*2001*	*2501*	*3001*	*3501*	*4001*	*10001*
parallel/quick_parallel	**44.79%**	**39.68%**	**44.%**	**41.03%**	**29.17%**	**25.71%**	**20.41%**	**11.24%**
parallel/serial	**-113%**	**-18.75%**	**50.88%**	**59.65%**	**58.54%**	**61.19%**	**61.76%**	**59.42%**
quick_parallel/serial	**-272%**	**-97%**	**12.28%**	**31.58%**	**41.46%**	**47.76%**	**51.96%**	**54.28%**

From the above data some interesting conclusions may be derived:

- *parallel* implementation is always faster that corresponding *quick_parallel* implementation of the underlying algorithm
- for reasonably large number of time steps utilized, *parallel* and *quick_parallel* implementations are faster that *serial* implementation of the underlying algorithm

Leisen-Reimer binomial tree, American Call

The American Call option is implemented by the following code portion of the original code

```
for ( j = n - 1; j >= 0; j-- )
 {
  for ( i = 0; i <= j; i++ )
```

```
    {
     AC[i][j] = max(S[i][j] - K, exp(-r*dt)*(p*AC[i][j + 1] +
                                             (1 - p)*AC[i + 1][j + 1]));
    }
  }
```

with **n** being the number of time steps utilized and **S** being the input binomial tree.
We will refer to the executable derived from this code as *serial* implementation.

The internal loop of the above code is parallel and may be easily parallelized using OpenMP - we will refer to the executable derived from that parallel version of code as *quick_parallel* implementation.

However we will create a better parallel code.

Perform Loop interchange of the above code getting the loop

```
 for ( i = n - 1; i >= 0; i-- )
  {
   for ( j = n - 1; j <= i; j-- )
    {
     AC[i][j] = max(S[i][j] - K, exp(-r*dt)*(p*AC[i][j + 1] +
                                             (1 - p)*AC[i + 1][j + 1]));
    }
  }
```

Now the internal loop is not parallel but it is reminiscent of the conditional stencil **dpl_stencil_max** and the technology, utilized by ***dpl*** for implementation of this stencil, may be applied to achieve the better performance for Leisen-Reimer implementation of the American Call option.
The major sources of optimization of the loop after interchange are:

- cache friendly memory access
- one parallel invocation for all the code
- the ability to create loop pipeline for the encompassing `for ( i = n - 1; i >= 0; i-- )` loop

Combined with the careful coding we obtain an executable which is refereed as *parallel* implementation.

The following timing improvement was observed on the **Intel® Core(TM) i5-9400** Processor with 6 threads. Each column represents the number of time steps utilized.
Each row represents improvement of the code of the first specified executable type over the code of the second executable type.

	1001	*1501*	*2001*	*2501*	*3001*	*3501*	*4001*	*10001*
parallel/quick_parallel	**36.79%**	**40.8%**	**42%**	**38.79%**	**32.25%**	**26.27%**	**25.45%**	**18.29%**
parallel/serial	**41.23%**	**69.17%**	**73.48%**	**74.64%**	**74.56%**	**73.64%**	**72.23%**	**69.54%**
quick_parallel/serial	**7.02%**	**47.92%**	**54.27%**	**58.57%**	**62.44%**	**64.44%**	**64.09%**	**62.73%**

From the above data some interesting conclusions may be derived:

- *parallel* implementation is always faster that corresponding *quick_parallel* implementation of the underlying algorithm
- *parallel* and *quick_parallel* implementations are always faster that *serial* implementation of the underlying algorithm

Crank-Nicolson finite difference method for option pricing

Here we will use Crank-Nicolson Finite Difference Method for Option Pricing to find the numeric solution to the Black, Scholes and Merton partial differential equation (see this for the description):

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

where:

t - a time in years, **t** = **0** being initial time ("now")
S(t) - the price of the underlying asset at time **t**
V(S,t) - the price of the option as a function of the underlying asset **S** at time **t**
r - the the annualized risk-free interest rate
σ - the volatility of the stock

We assume the time is running from "now" (**0**) to some specified value **YearsToMaturity** the price of underlying asset is running from **0** to some specified value **MaxPrice**.

Divide the (**t**, **S**) plane into a grid of **N**+**1** x **M**+**1** equally spaced points
$t_i = \Delta t\ i,\ i=0,...,N$
$S_j = \Delta S\ j,\ j = 0,...,M$
where
Δt = **YearsToMaturity**/**N**
ΔS = **MaxPrice**/**M**
We denote the value of the derivative at time t_i when the underlying asset has value S_j as
$V_{i,j} = V(i\,\Delta t, j\,\Delta S) = V(t_i, S_j)$ for $i = 0,...,N;\ j = 0,...,M$

Crank-Nicolson method uses the following discretization of the above the Black, Scholes and Merton partial differential equation
$-a_j V_{i-1,j-1} + (1 - b_j)V_{i-1,j} - c_j V_{i-1,j+1} = a_j V_{i,j-1} + (1 + b_j)V_{i,j} + c_j V_{i,j+1}$

for $\mathbf{i} = 1,...,N$; $\mathbf{j} = 1,...,M-1$

where

$\mathbf{a_j} = \Delta t(\sigma^2 j^2 - rj)*0.25$

$\mathbf{b_j} = -\Delta t(\sigma^2 j^2 + r)*0.5$

$\mathbf{c_j} = \Delta t(\sigma^2 j^2 + rj)*0.25$

For a given **StrikePrice**, to calculate European Put option, we establish the following boundary conditions:

$V_{i,0} = StrikePrice*\exp(^{-r(N-i)\Delta t})$ for $\mathbf{i} = 0,1,...,N$

$V_{i,M} = 0$ for $\mathbf{i} = 0,1,...,N$

$V_{N,j} = \max(StrikePrice - j\Delta S, 0)$ for $\mathbf{j} = 0,1,...,M$

The discretization cited earlier can be rewritten as:

$(1 - b_j)V_{i-1,j} = a_j V_{i-1,j-1} + c_j V_{i-1,j+1} + a_j V_{i,j-1} + (1 + b_j)V_{i,j} + c_j V_{i,j+1}$

for $\mathbf{i} = 1,...,N$; $\mathbf{j} = 1,...,M-1$

or

$V_{i-1,j} = A_j V_{i-1,j-1} + B_j$

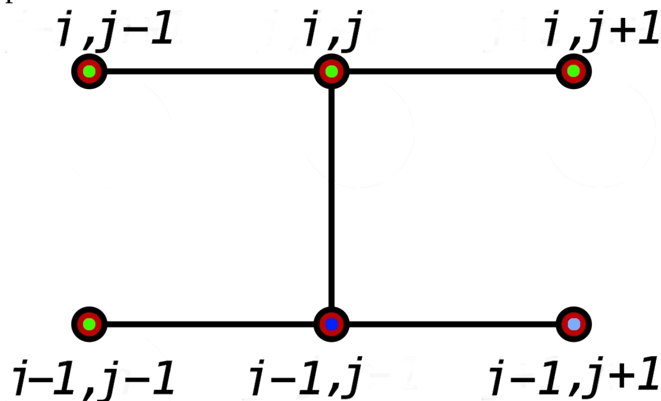
for $\mathbf{i} = 1,...N$; $\mathbf{j} = 1,...,M-1$

where

$A_j = a_j/(1 - b_j)$

$B_j = (c_j V_{i-1,j+1} + a_j V_{i,j-1} + (1 + b_j)V_{i,j} + c_j V_{i,j+1})/(1 - b_j)$

The newly created expression is a stencil for which the technology provided by ***dpl*** enables parallel execution.



We execute these stencils for **i** starting with **N** till **1**. At every given step:

- $V_{i-1,j-1}$ is defined by the established boundary condition (for **j** being **1**) or calculated at the previous step of the stencil (for other values of **j**)
- $V_{i-1,j+1}$ is defined only for **j** being **M**-1 (as the established boundary condition); for other values of **j** the currently existing value of $V_{i-1,j+1}$ is used; that requires the initialization of all the elements of **V** (except the boundary described earlier) to some reasonable initial values and iterations of the algorithm till the desirable precision has been reached
- $V_{i,j-1}$, $V_{i,j}$. $V_{i,j+1}$ are defined by the the established boundary conditions (for **i** being **N**) or calculated at the previous iterations of the proposed algorithm (for other values of **i**)

The following timing improvement was observed on the **Intel® Core(TM) i5-9400** Processor with 6 threads.
We used the square grid - **N** being equal to **M**. Each column represents the value of **N** - number of time steps utilized. The row represents improvement of the *parallel* version of the described algorithm over it *serial* version.

	200	***250***	***300***	***350***	***400***	***450***	***500***
parallel/serial	**-6.14%**	**17.75%**	**28%**	**37%**	**41%**	**50%**	**54%**

Heston stochastic volatility model

Heston stochastic volatility model is an extension of the Black-Scholes model that doesn't assume a volatility to be constant.

The following shows the results of optimization (parallelization) for calculation of a European Call option using Heston model for a discretisation technique known as "Full Truncation Euler Discretisation, coupled with Monte Carlo simulation", see this for the theoretical reference.

The implemented algorithm is as follows:

Generation of correlated normally distributed random numbers sequences

We set **SpotNormalRandom** array to a sequence of standard normal random numbers and set **VolatilityNormalRandom** to a sequence of standard normal random numbers correlated with **SpotNormalRandom**.

To generate a sequence **VolatilityNormalRandom** of standard normal random numbers:
we first generated a sequence of pairs of an uncorrelated random variables in the range]**0.**,**1.**[using random number generator provided by the Dalsoft's Random Testing (***drt***) package (see this for more information)
then, for every pair of the generated random numbers, we applied the Box–Muller transform generating pairs of independent, standard, normally distributed random numbers - see this for explanation.

To make **VolatilityNormalRandom** to be correlated with **SpotNormalRandom** we used Cholesky decomposition (as it is used in the Monte Carlo method for simulating systems with multiple correlated variables) - the algorithm is described here (also see this for additional discussion of the used algorithm); the algorithm is using **ρ** - correlation of asset and volatility, which is a given input parameter.

The generation of correlated normally distributed random numbers sequences was fully parallelized.

Calculation of the volatility price **VolatilityPrices** values from the **VolatilityNormalRandom** vector generated in the previous step

The following code was implemented for this calculation:

```
for ( i = 1; i < NumberOfPoints; i++ )
 {
 VolMax = max( VolatilityPrices[i-1], 0.0 );

 VolatilityPrices[i] = VolatilityPrices[i-1] + κ*Δt*( θ - VolMax ) +
                 ξ*sqrt( VolMax*Δt )*VolatilityNormalRandom[i-1];
```

where
κ - mean-reversion rate
θ - long run average volatility
ξ - the volatility of the volatility
Δ**t** - time derivative discretisation

We failed to parallelize the calculation of volatility prices. However, the above code was greatly optimized by applying the optimization technology utilized by the ***dpl***, e.g. techniques for hiding memory latencies.

Calculation of the spot price **SpotPrices** values from the **SpotNormalRandom** and **VolatilityPrices** vectors generated in the previous steps

The code utilized:

```
for ( i = 1; i < NumberOfPoints; i++ )
 {
 VolMax = max( VolatilityPrices[i-1], 0.0 );

 SpotPrices[i] = SpotPrices[i-1]*exp( ( r - 0.5*VolMax )*Δt +
         sqrt( VolMax*Δt )*SpotNormalRandom[i-1] );
 }
```

where
r - risk-free rate
Δ**t** - time derivative discretization

The above code is a stencil supported by ***dpl*** and was fully parallelized.

The following timing improvement was observed on the **Intel® Core(TM) i5-9400** Processor with 6 threads.
We run every simulation for **100000** iterations.
Each column represents the value of **NumberOfPoints** - number of time steps utilized.
The row represents improvement of the optimized (*parallel*) version of the described algorithm over it *serial* version.

	500	***1000***	***1500***	***2000***	***2500***	***3000***	***3500***
parallel/serial	**61.79%**	**65.11%**	**66.68%**	**66.87%**	**66.91%**	**67.73%**	**68.54%**