# A Cloud Based Super-Optimization Method to Parallelize the Sequential Code's Nested Loops

Amin Majd, Mohammad Loni, Golnaz Sahebi, Masoud Daneshtalab,
Elena Troubitsyna
Åbo Akademi, University of Turku, KTH, Mälardalen University

## Motivation

- Multi-core processors offer powerful computing capacity
- However, parallel programming is complex for ordinary software engineers
- Alternative solution to dynamic optimization techniques provided by super compilers
- Super compilers are dynamic code optimizers that run the independent parts of the code on available cores in parallel
- The idea: create an automated parallilisation method hosted in cloud and provide parallelisation-as-a-service

## Overview of the parallelization approach

Overall aim: to convert nested loops into parallel structures

Steps:

1. Creating a table to save loop data dependencies
2. Tiling the iteration space for achieving better parallelization performance.
   - A tile is a set of loop iterations running on the same processor.
3. Generating parallel code corresponding to the structure and size of produced tiles automatically for the iteration space produced in step 2.
4. Scheduling the tiling space of step 3.

## Cloud-based solution

- A programmer sends the sequential codes to the cloud and then receives the parallel version of the code.
- No need to understand parallel computing to achieve a high utilization of many cores.
- The potential users include sequential code programmers, programmers not familiar with parallel programming or supercompliers or cannot afford supercompilers.

## Principles of loop parallelisation

- The data dependency exists between two adjacent data references if both references access the same memory location, and at least one of them is a write access

*for (p = 6; p < q; p++ )*

*y[p] = y[p-1] + I ;*

*p=6 p=7 p=8*

*y[6] = y[5] + I y[7] = y[6] + I y[8] = y[7] + I*

## Principles of loop parallelisation

- The data dependency exists between two adjacent data references if both references access the same memory location, and at least one of them is a write access

*for (p = 6; p < q; p++ )*

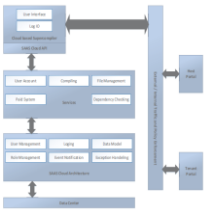
*y[p] = y[p] + I ;*

*p=6 p=7 p=8*

*y[6] = y[6] + I y[7] = y[7] + I y[8] = y[8] + I*

- Diophantine equations are used to identify data dependencies in the code
  - Diophantine equations are linear equations usually with two or more unknown variables that accept only the integer values as solutions
- Solving Diophantine equations is NP-hard constraint solving problem

## System architecture



- A cloud service is simulated by connecting eight PCs via a switch.
- VM represents a cloud center
- The client PC sends serial code, which contains the nested loop, to the cloud center,
- Cloud center analyzes it to find any data dependency using the Diophantine equations
- It selects independent data to create a parallel code

## Genetic Algorithms

- GA is an iterative population-based solution space exploration approach
- GA mimic the process of natural selection and evolution
- Solutions are encoded as chromosomes
- The initial population is generated at random
- Fitness function is introduced to evaluate a quality of a solution
- The operations of selection, crossover and mutation are iteratively applied to improve the characteristics of the population and find a (quasi) optimal solution
- The operations are repeated until a predefined criteria is satisfied or execution time elapses

## GA for parallelization optimization (1/2)

- The Tile-to-Processor assignment matrix is used for encoding the chromosomes.
- This matrix shows the processor assignment to the available tiles
- *Chromosome is a scheduling for all the jobs*
- *Initial Population* consists of randomly generated solutions
- *Fitness Function* compare different scheduling that satisfy problem constraints

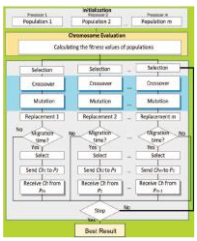
  *Fitness (s) = 1/makespan(s), makespan (s)= max (CompilationTime (J)),*

  *J* tiled iteration space task scheduling.
  - *makespan (s)* represents the total execution time of all scheduled tiles

## GA for parallelization optimization (2/2)

- *Selection Operator:* the Roulette wheel and Tournament selection for choosing parent chromosome.
- *Crossover Operator:* uniform crossover method is used for crossing the selected pairs with each other.
- *Mutation Operator : bit-flipping*
- *Migration Operator:* after a predefined number of iterations, all the populations share their best chromosomes among each other.

## Parallel implementation of GA




## Examples of parallilisation and experiments

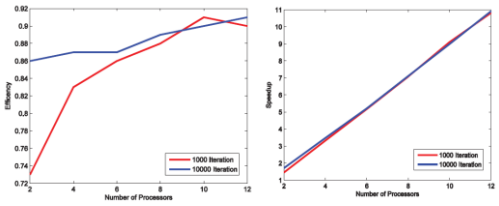
```
1.  for (i = 0; i < n; i++)
2.      A[i] = 2 * A[i];
```

```
1.  MPI_Init ( NULL,  NULL ) ;
2.  int size ;
3.  MPI_Comm_size ( MPI_COMM_WORLD,  &size ) ;
4.  int rank ;
5.  MPI_Comm_rank ( MPI_COMM_WORLD, & rank )
    ;
6.  int p;
7.  p = n / size;
8.  if ( rank != ( size - 1 ) )
9.      for ( i = rank * p ; i < ( rank * p ) +  p ;  i++ )
10.         A[i] = 2 * A[i];
11. else
12.     for (i = rank * p ; i < n ; i++ )
13.         A[i] = 2 * A[i];
14. MPI_Finalize ();
```

```
1.  for (i = 0; i < n; i++)
2.      for (j = 0; j < m; j++)
3.          A[i][j] = A[i][j] + 1;
```

```
1.  MPI_Init (NULL, NULL);
2.  int size;
3.  MPI_Comm_size (MPI_COMM_WORLD, & size);
4.  int rank;
5.  MPI_Comm_rank (MPI_COMM_WORLD, & rank);
6.  int p;
7.  p = n / size;
8.  if ( rank != ( size -1 ) )
9.      for (i = rank * p; i< (rank * p) + p; i++)
10.         for (j = 0; j < m; j++)
11.             A[i][j] = A[i] + 1;
12. else
13.     for (i = rank * p; i< n; i++)
14.         for (j = 0; j <m; j++)
15.             A[i][j] = A[i] + 1;
16. MPI_Finalize ();
```

## Examples of parallilisation and experiments




## Conclusions

- We have proposed an alternative approach to parallilisation of sequential programs
- It relies on cloud and optimization
- Parallel code is delivered as a service
- We aimed at alleviating problems associated with parallel programming and costs of super-compilers
- Future work: experimenting with larger and more complex programs