Towards Parallel Constraint-Based Local Search with the X10 Language

Danny Munera , Daniel Diaz and Salvador Abreu

University of Paris 1-Sorbonne, France
Universidade de Evora and CENTRIA, Portugal

INAP 2013
Kiel, Germany, September 2013

Université Paris 1 – Panthéon - Sorbonne –

Centre de Recherche en Informatique

Universidade de Évora

Agenda

- Context
- X10 programming language
- Adaptive Search
  - Parallel Implementation
- Experimentation Results
- Conclusion and Future Work

Agenda

- Context
- X10 programming language
- Adaptive Search
  - Parallel Implementation
- Experimentation Results
- Conclusion and Future Work

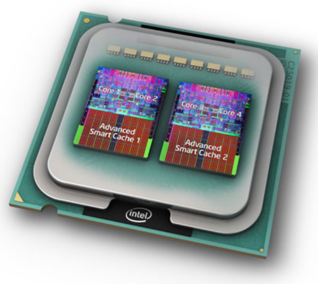
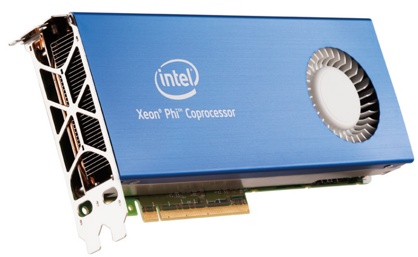
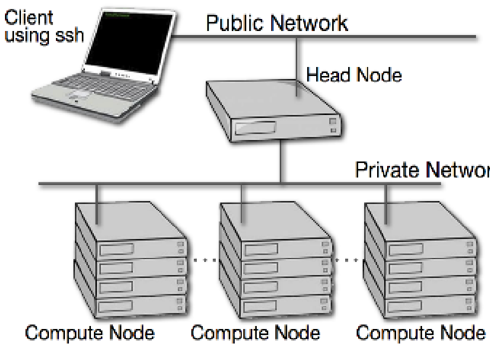
Constraint Programming

- Constraint Programming
  - Successfully used to model **Real-Life Problems**
    - Planning
    - Resource allocation
    - Scheduling
    - Product line modeling

Constraint Programming - Solving

- Exhaustively by complete methods:
  - Can find all solutions
  - Exponential growth of Search Space
    - **Magic Square 15 x 15**
- Completeness and resorting to (meta-) heuristics
  - Can attack problems out of the scope of complete solvers
  - Local search method can easily solve **MS 100 x 100 problem**

How to improve solving performance?

Multi-Core	Many-Core	GPGPUs	Cluster and Grid
			

C++ pthreads

OpenMP

NVIDIA CUDA

MPI

X10

Our Experimentation

- Contraint-Based Local Search Algorithm:
  - Adaptive Search
- Different Parallel Implementations:
  - **Functional Parallelism**
  - **Data Parallelism**
- PGAS Model - X10

Agenda

- Context
- X10 programming language
- Adaptive Search
  - Parallel Implementation
- Experimentation Results
- Conclusion and Future Work

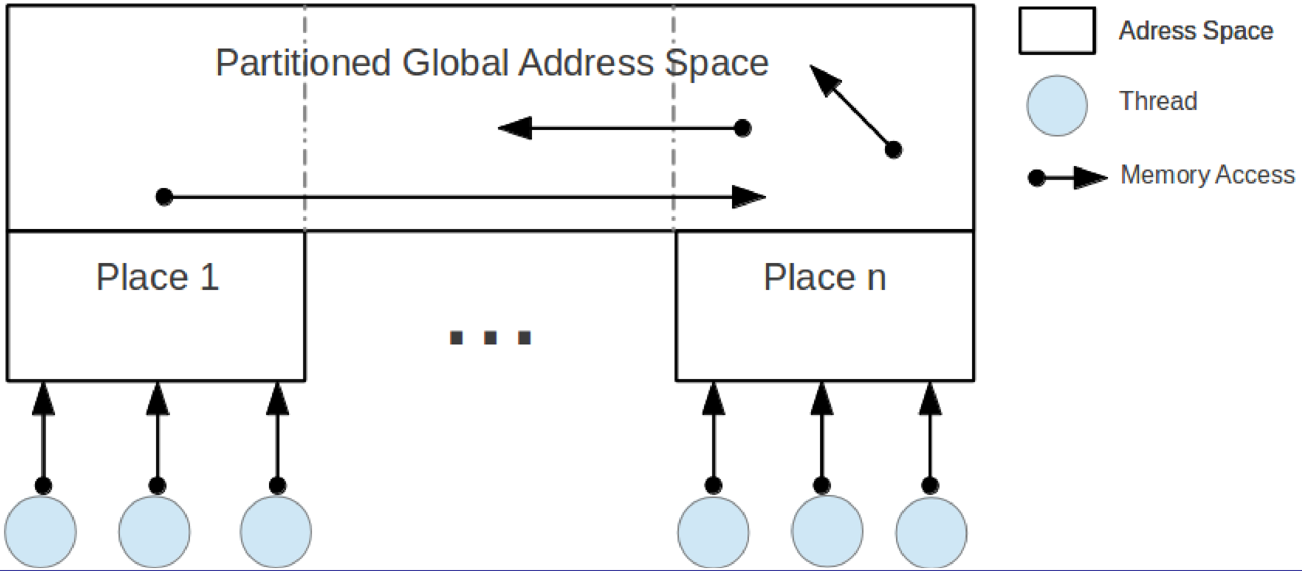
X10 Programming Language

- **General-purpose language developed by IBM**
  - Asynchronous PGAS (APGAS).
    - Extends the PGAS model making it flexible, even in non-HPC platforms
  - **Support different levels of concurrency** with simple language constructs.
  - **Java**-like language
  - **Single programming model for hetereogeneity**

X10 in a Nutshel

- Two main abstractions
  - **Places:** virtual shared-memory process.
    - Coherent portion of the address space together with threads (activities).
    - `at (p) S`
  - **Activities:**
    - Single thread that perform computation within a place
    - `async (S)`

http://x10-lang.org



Agenda

- Context
- X10 programming language
- Adaptive Search
  - Parallel Implementations
- Experimentation Results
- Conclusion and Future Work

The Adaptive Search Method

- Generic, domain-independent constraint-based **Local Search** method.
- Takes advantage of the **CSP formulation** and makes it possible to structure the problem in terms of variables and constraints.
- **Adaptive memory** inspired in Tabu Search.

The Adaptive Search Method - Permutation

```
repeat
    Compute a random assignment A of variables in V
    repeat
        Compute errors constraints in C
        Select variable X with highest error: MaxV
        Select the move with best cost from X: MinConflictV
        if no improvement move exists then
            mark X as Tabu for T iterations
            if number of variables marked Tabu >= RL then
                randomly reset some variables in V
            end if
        else
            swap( MaxV , MinConflictV )
            if cost(A) < Opt_Cost then
                Opt_Sol = A
                Opt_Cost = cost(A)
            end if
        end if
    until Opt_Cost = 0 (solution found) or Iteration >= MI
until Opt_Cost = 0 (solution found) or Restart >= MR
Output ( Opt_Sol, Opt_Cost )
```

The Adaptive Search Method - Permutation

```
repeat
    Compute a random assignment A of variables in V
    repeat
        Compute errors constraints in C
        Select variable X with highest error: MaxV
        Select the move with best cost from X: MinConflictV
        if no improvement move exists then
            mark X as Tabu for T iterations
            if number of variables marked Tabu >= RL then
                randomly reset some variables in V
            end if
        else
            swap( MaxV , MinConflictV )
            if cost(A) < Opt_Cost then
                Opt_Sol = A
                Opt_Cost = cost(A)
            end if
        end if
    until Opt_Cost = 0 (solution found) or Iteration >= MI
until Opt_Cost = 0 (solution found) or Restart >= MR
Output ( Opt_Sol, Opt_Cost )
```

The Adaptive Search Method - Permutation

```
repeat
    Compute a random assignment A of variables in V
    repeat
        Compute errors constraints in C
        Select variable X with highest error: MaxV
        Select the move with best cost from X: MinConflictV
        if no improvement move exists then
            mark X as Tabu for T iterations
            if number of variables marked Tabu >= RL then
                randomly reset some variables in V
            end if
        else
            swap( MaxV , MinConflictV )
            if cost(A) < Opt_Cost then
                Opt_Sol = A
                Opt_Cost = cost(A)
            end if
        end if
    until Opt_Cost = 0 (solution found) or Iteration >= MI
until Opt_Cost = 0 (solution found) or Restart >= MR
Output ( Opt_Sol, Opt_Cost )
```

Main loop

The Adaptive Search Method - Permutation

```
repeat
    Compute a random assignment A of variables in V
    repeat
        Compute errors constraints in C
        Select variable X with highest error: MaxV
        Select the move with best cost from X: MinConflictV
        if no improvement move exists then
            mark X as Tabu for T iterations
            if number of variables marked Tabu >= RL then
                randomly reset some variables in V
            end if
        else
            swap( MaxV , MinConflictV )
            if cost(A) < Opt_Cost then
                Opt_Sol = A
                Opt_Cost = cost(A)
            end if
        end if
    until Opt_Cost = 0 (solution found) or Iteration >= MI
until Opt_Cost = 0 (solution found) or Restart >= MR
Output ( Opt_Sol, Opt_Cost )
```

Main loop

The Adaptive Search Method - Permutation

```
repeat
    Compute a random assignment A of variables in V
    repeat
        Compute errors constraints in C
        Select variable X with highest error: MaxV
        Select the move with best cost from X: MinConflictV
        if no improvement move exists then
            mark X as Tabu for T iterations
            if number of variables marked Tabu >= RL then
                randomly reset some variables in V
            end if
        else
            swap( MaxV , MinConflictV )
            if cost(A) < Opt_Cost then
                Opt_Sol = A
                Opt_Cost = cost(A)
            end if
        end if
    until Opt_Cost = 0 (solution found) or Iteration >= MI
until Opt_Cost = 0 (solution found) or Restart >= MR
Output ( Opt_Sol, Opt_Cost )
```

Main loop

The Adaptive Search Method - Permutation

```
repeat
    Compute a random assignment A of variables in V
    repeat
        Compute errors constraints in C
        Select variable X with highest error: MaxV
        Select the move with best cost from X: MinConflictV
        if no improvement move exists then
            mark X as Tabu for T iterations
            if number of variables marked Tabu >= RL then
                randomly reset some variables in V
            end if
        else
            swap( MaxV , MinConflictV )
            if cost(A) < Opt_Cost then
                Opt_Sol = A
                Opt_Cost = cost(A)
            end if
        end if
    until Opt_Cost = 0 (solution found) or Iteration >= MI
until Opt_Cost = 0 (solution found) or Restart >= MR
Output ( Opt_Sol, Opt_Cost )
```

Main loop

The Adaptive Search Method - Permutation

```
repeat
    Compute a random assignment A of variables in V
    repeat
        Compute errors constraints in C
        Select variable X with highest error: MaxV
        Select the move with best cost from X: MinConflictV
        if no improvement move exists then
            mark X as Tabu for T iterations
            if number of variables marked Tabu >= RL then
                randomly reset some variables in V
            end if
        else
            swap( MaxV , MinConflictV )
            if cost(A) < Opt_Cost then
                Opt_Sol = A
                Opt_Cost = cost(A)
            end if
        end if
    until Opt_Cost = 0 (solution found) or Iteration >= MI
until Opt_Cost = 0 (solution found) or Restart >= MR
Output ( Opt_Sol, Opt_Cost )
```

Main loop

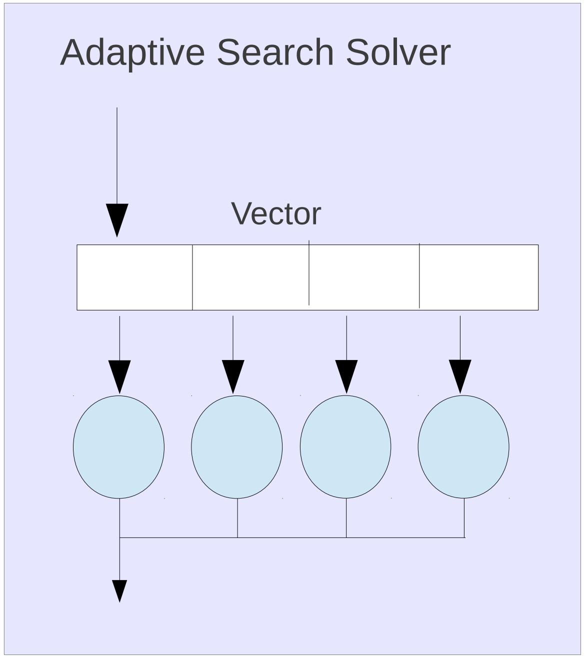
Agenda

- Context
- X10 programming language
- Adaptive Search
  - Parallel Implementations
- Experimentation Results
- Conclusion and Future Work

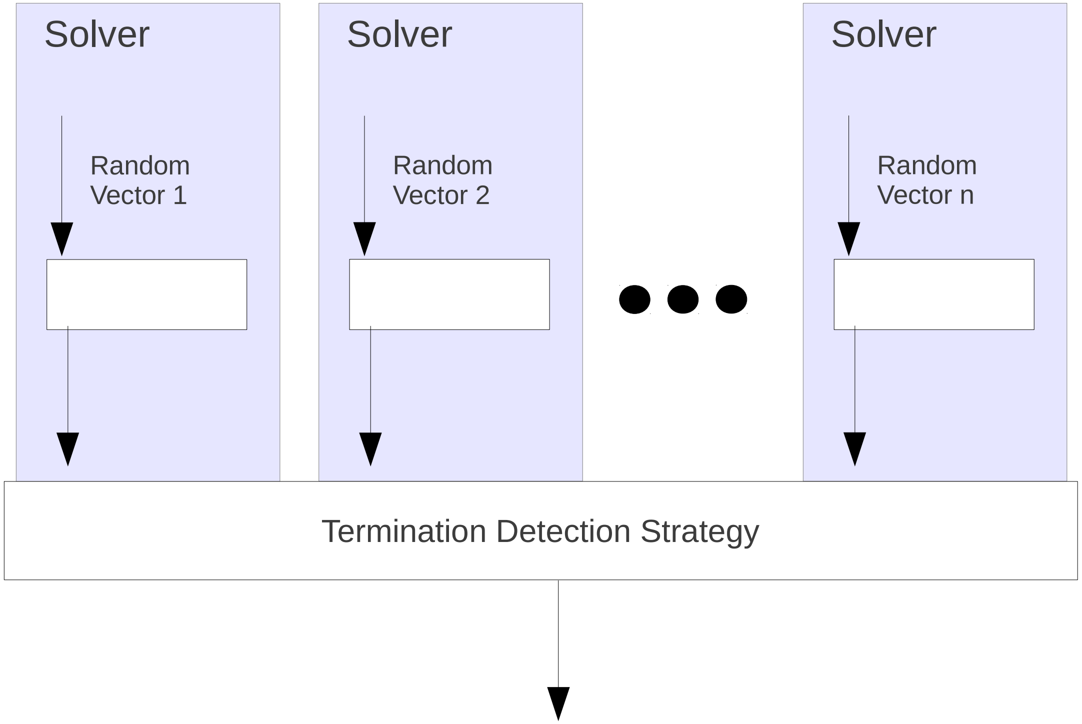
The Adaptive Search Method

- Sources of parallelism

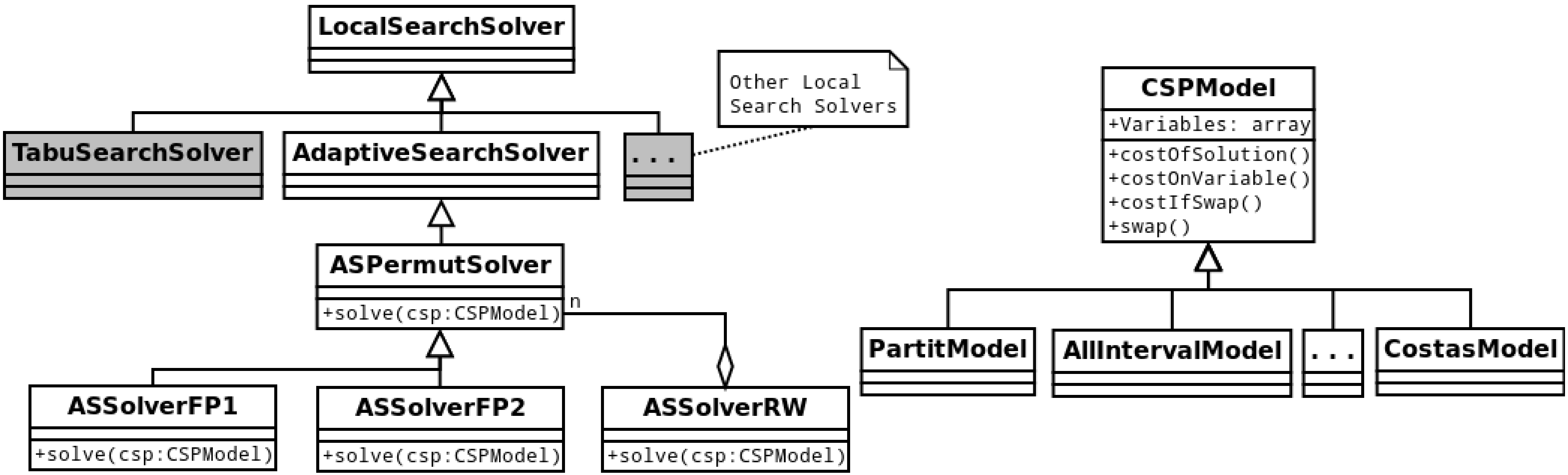
Functional Parallelism



Data Parallelism



The Adaptive Search Method X10 Implementation



The Adaptive Search Method X10 Implementation

```
class ASPermutSolver {
  var totalCost: Int;
  var maxI : Int;
  var minJ : Int;

  public def solve (csp : CSPModel) : Int {
    ... local variables ...
    csp.initialize();
    totalCost = csp.costOfSolution();
    while (totalCost != 0) {
      ... restart code ...
      maxI = selectVarHighCost (csp);
      minJ = selectVarMinConflict (csp);
      ... local min tabu list, reset code ...
      csp.swapVariables (maxI, minJ);
      totalCost = csp.costOfSolution ();
    }
    return totalCost;
  }
}
```

Main Loop

The Adaptive Search Method X10 Implementation

```
class ASPermutSolver {
  var totalCost: Int;
  var maxI : Int;
  var minJ : Int;

  public def solve (csp : CSPModel) : Int {
    ... local variables ...
    csp.initialize();
    totalCost = csp.costOfSolution();
    while (totalCost != 0) {
      ... restart code...
      maxI = selectVarHighCost (csp);
      minJ = selectVarMinConflict (csp);
      ... local min tabu list, reset code ...
      csp.swapVariables (maxI, minJ);
      totalCost = csp.costOfSolution ();
    }
    return totalCost;
  }
}
```

Main Loop

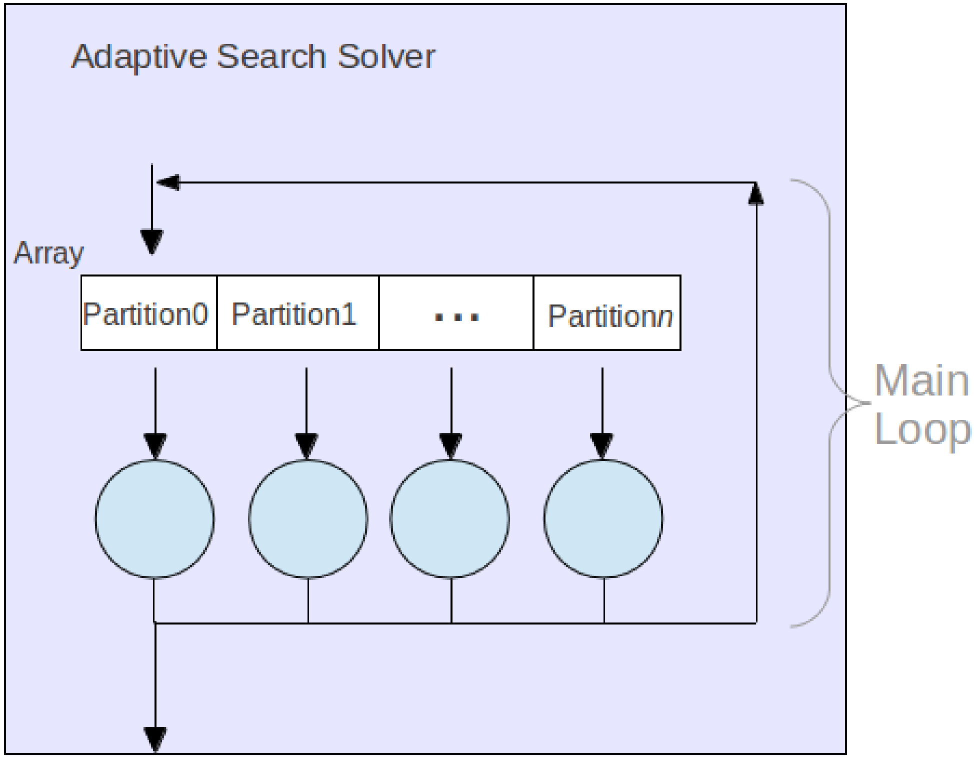
Adaptive Search – X10 Functional Parallelism

Sequential Implementation

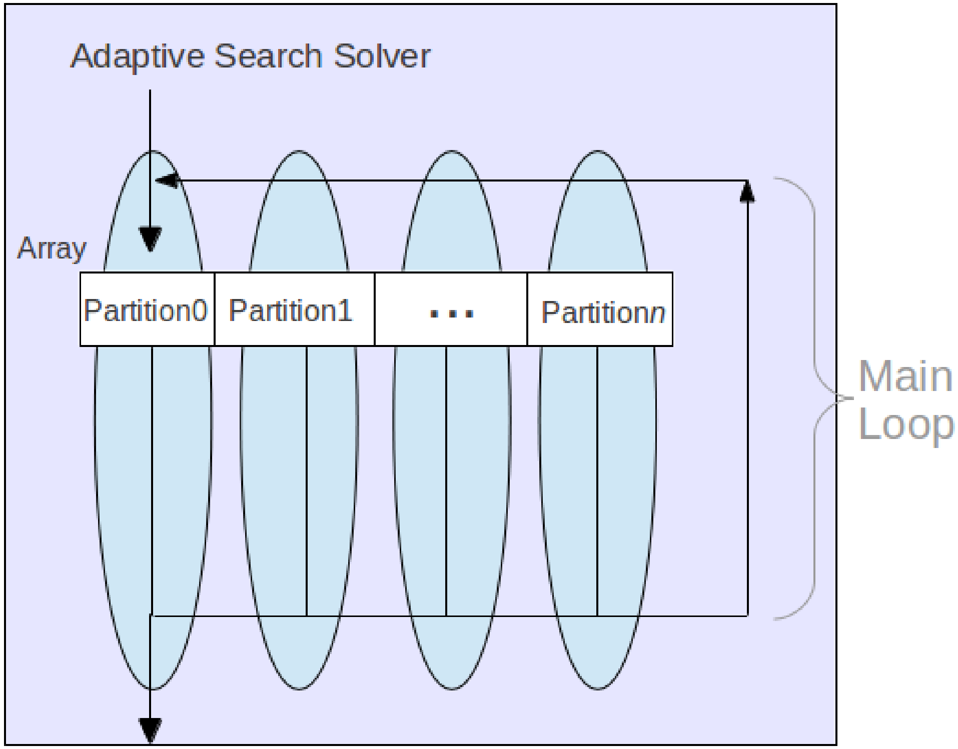
```
public def selectVarHighCost( csp : CSPModel ) : Int {
    . . . local variables . . .
    // main loop: go through each variable in the CSP
    for (i = 0; i < size; i++) {
        . . . count marked variables . . .
        cost = csp.costOnVariable (i);
        . . . select the highest cost . . .
    }
    return maxI; // index of the highest cost
}
```

Adaptive Search – X10 Functional Parallelism

First Approach



Second Approach



Adaptive Search – X10 Functional Parallelism

First approach to functional parallelism

```
public def selectVarHighCost (csp : CSPModel) : Int {
    // Initialization of Global variables
    var partition : Int = csp.size/THNUM;
    finish for(th in 1..THNUM){
        async{
            for (i = ((th-1)*partition); i < th*partition; i++){
                . . . calculate individual cost of each variable . . .
                . . . save variable with higher cost . . .
            }
        }
    }
    . . . terminate function: merge solutions . . .
    return maxI; // Index of the higher cost
}
```

Adaptive Search – X10 Functional Parallelism

First approach to functional parallelism

```
public def selectVarHighCost (csp : CSPModel) : Int {
    // Initialization of Global variables
    var partition : Int = csp.size/THNUM;
    finish for(th in 1..THNUM){
        async{
            for (i = ((th-1)*partition); i < th*partition; i++){
                . . . calculate individual cost of each variable . . .
                . . . save variable with higher cost . . .
            }
        }
    }
    . . . terminate function: merge solutions . . .
    return maxI; // Index of the higher cost
}
```

Adaptive Search – X10 Functional Parallelism

First approach to functional parallelism

```
public def selectVarHighCost (csp : CSPModel) : Int {
    // Initialization of Global variables
    var partition : Int = csp.size/THNUM;
    finish for(th in 1..THNUM){
        async{
            for (i = ((th-1)*partition); i < th*partition; i++){
                . . . calculate individual cost of each variable . . .
                . . . save variable with higher cost . . .
            }
        }
    }
    . . . terminate function: merge solutions . . .
    return maxI; // Index of the higher cost
}
```

Adaptive Search – X10 Functional Parallelism

First approach to functional parallelism

```
public def selectVarHighCost (csp : CSPModel) : Int {
    // Initialization of Global variables
    var partition : Int = csp.size/THNUM;
    finish for(th in 1..THNUM){
        async{
            for (i = ((th-1)*partition); i < th*partition; i++){
                . . . calculate individual cost of each variable . . .
                . . . save variable with higher cost . . .
            }
        }
    }
    . . . terminate function: merge solutions . . .
    return maxI; // Index of the higher cost
}
```

Adaptive Search – X10 Functional Parallelism

First approach to functional parallelism

```
public def selectVarHighCost (csp : CSPModel) : Int {
    // Initialization of Global variables
    var partition : Int = csp.size/THNUM;
    finish for(th in 1..THNUM){
        async{
            for (i = ((th-1)*partition); i < th*partition; i++){
                . . . calculate individual cost of each variable . . .
                . . . save variable with higher cost . . .
            }
        }
    }
    . . . terminate function: merge solutions . . .
    return maxI; // Index of the higher cost
}
```

Second approach to functional parallelism

```
public class ASSolverFP1 extends ASPermutSolver{
    val computeInst : Array[ComputePlace];
    var startBarrier : ActivityBarrier;
    var doneBarrier : ActivityBarrier;
    public def solve(csp : CSPModel):Int{
        for(var th : Int = 1; th <= THNUM ; th++)
            computeInst(th) = new ComputePlace(th , csp);
        for(id in computeInst)
            async computeInst(id).run();
        while(total cost!=0){
            . . . restart code . . .
            for(id in computeInst)
                computeInst(id).activityToDo = SELECVARHIGHCOST;
            startBarrier.wait(); // send start signal
            // activities working...
            doneBarrier.wait(); // work ready
            maxI=terminateSelVarHighCost();
            . . . local min tabu list, reset code . . .
        }
        // Finish activities
        for(id in computeInst)
            computeInst(id).activityToDo = FINISH;
        startBarrier.wait();
        doneBarrier.wait();
```

Second approach to functional parallelism

```
public class ASSolverFP1 extends ASPermutSolver{
    val computeInst : Array[ComputePlace];
    var startBarrier : ActivityBarrier;
    var doneBarrier : ActivityBarrier;
    public def solve(csp : CSPModel):Int{
        for(var th : Int = 1; th <= THNUM ; th++)
            computeInst(th) = new ComputePlace(th , csp);
        for(id in computeInst)
            async computeInst(id).run();
        while(total cost!=0){
            . . . restart code . . .
            for(id in computeInst)
                computeInst(id).activityToDo = SELECVARHIGHCOST;
            startBarrier.wait(); // send start signal
            // activities working...
            doneBarrier.wait(); // work ready
            maxI=terminateSelVarHighCost();
            . . . local min tabu list, reset code . . .
        }
        // Finish activities
        for(id in computeInst)
            computeInst(id).activityToDo = FINISH;
        startBarrier.wait();
        doneBarrier.wait();
```

Second approach to functional parallelism

```
public class ASSolverFP1 extends ASPermutSolver{
    val computeInst : Array[ComputePlace];
    var startBarrier : ActivityBarrier;
    var doneBarrier : ActivityBarrier;
    public def solve(csp : CSPModel):Int{
        for(var th : Int = 1; th <= THNUM ; th++)
            computeInst(th) = new ComputePlace(th , csp);
        for(id in computeInst)
            async computeInst(id).run();
        while(total cost!=0){
            . . . restart code . . .
            for(id in computeInst)
                computeInst(id).activityToDo = SELECVARHIGHCOST;
            startBarrier.wait(); // send start signal
            // activities working...
            doneBarrier.wait(); // work ready
            maxI=terminateSelVarHighCost();
            . . . local min tabu list, reset code . . .
        }
        // Finish activities
        for(id in computeInst)
            computeInst(id).activityToDo = FINISH;
        startBarrier.wait();
        doneBarrier.wait();
```

Second approach to functional parallelism

```
public class ASSolverFP1 extends ASPermutSolver{
    val computeInst : Array[ComputePlace];
    var startBarrier : ActivityBarrier;
    var doneBarrier : ActivityBarrier;
    public def solve(csp : CSPModel):Int{
        for(var th : Int = 1; th <= THNUM ; th++)
            computeInst(th) = new ComputePlace(th , csp);
        for(id in computeInst)
            async computeInst(id).run();
        while(total cost!=0){
            . . . restart code . . .
            for(id in computeInst)
                computeInst(id).activityToDo = SELECVARHIGHCOST;
            startBarrier.wait(); // send start signal
            // activities working...
            doneBarrier.wait(); // work ready
            maxI=terminateSelVarHighCost();
            . . . local min tabu list, reset code . . .
        }
        // Finish activities
        for(id in computeInst)
            computeInst(id).activityToDo = FINISH;
        startBarrier.wait();
        doneBarrier.wait();
```

Main Loop

Second approach to functional parallelism

```
public class ASSolverFP1 extends ASPermutSolver{
    val computeInst : Array[ComputePlace];
    var startBarrier : ActivityBarrier;
    var doneBarrier : ActivityBarrier;
    public def solve(csp : CSPModel):Int{
        for(var th : Int = 1; th <= THNUM ; th++)
            computeInst(th) = new ComputePlace(th , csp);
        for(id in computeInst)
            async computeInst(id).run();
        while(total cost!=0){
            . . . restart code . . .
            for(id in computeInst)
                computeInst(id).activityToDo = SELECVARHIGHCOST;
            startBarrier.wait(); // send start signal
            // activities working...
            doneBarrier.wait(); // work ready
            maxI=terminateSelVarHighCost();
            . . . local min tabu list, reset code . . .
        }
        // Finish activities
        for(id in computeInst)
            computeInst(id).activityToDo = FINISH;
        startBarrier.wait();
        doneBarrier.wait();
```

Main Loop

Second approach to functional parallelism

```
public class ASSolverFP1 extends ASPermutSolver{
    val computeInst : Array[ComputePlace];
    var startBarrier : ActivityBarrier;
    var doneBarrier : ActivityBarrier;
    public def solve(csp : CSPModel):Int{
        for(var th : Int = 1; th <= THNUM ; th++)
            computeInst(th) = new ComputePlace(th , csp);
        for(id in computeInst)
            async computeInst(id).run();
        while(total cost!=0){
            . . . restart code . . .
            for(id in computeInst)
                computeInst(id).activityToDo = SELECVARHIGHCOST;
            startBarrier.wait(); // send start signal
            // activities working...
            doneBarrier.wait(); // work ready
            maxI=terminateSelVarHighCost();
            . . . local min tabu list, reset code . . .
        }
        // Finish activities
        for(id in computeInst)
            computeInst(id).activityToDo = FINISH;
        startBarrier.wait();
        doneBarrier.wait();
```

Main Loop

Second approach to functional parallelism

```
public class ASSolverFP1 extends ASPermutSolver{
    val computeInst : Array[ComputePlace];
    var startBarrier : ActivityBarrier;
    var doneBarrier : ActivityBarrier;
    public def solve(csp : CSPModel):Int{
        for(var th : Int = 1; th <= THNUM ; th++)
            computeInst(th) = new ComputePlace(th , csp);
        for(id in computeInst)
            async computeInst(id).run();
        while(total cost!=0){
            . . . restart code . . .
            for(id in computeInst)
                computeInst(id).activityToDo = SELECVARHIGHCOST;
            startBarrier.wait(); // send start signal
            // activities working...
            doneBarrier.wait(); // work ready
            maxI=terminateSelVarHighCost();
            . . . local min tabu list, reset code . . .
        }
        // Finish activities
        for(id in computeInst)
            computeInst(id).activityToDo = FINISH;
        startBarrier.wait();
        doneBarrier.wait();
```

Main Loop

Second approach to functional parallelism

```
public class ASSolverFP1 extends ASPermutSolver{
    val computeInst : Array[ComputePlace];
    var startBarrier : ActivityBarrier;
    var doneBarrier : ActivityBarrier;
    public def solve(csp : CSPModel):Int{
        for(var th : Int = 1; th <= THNUM ; th++)
            computeInst(th) = new ComputePlace(th , csp);
        for(id in computeInst)
            async computeInst(id).run();
        while(total cost!=0){
            . . . restart code . . .
            for(id in computeInst)
                computeInst(id).activityToDo = SELECVARHIGHCOST;
            startBarrier.wait(); // send start signal
            // activities working...
            doneBarrier.wait(); // work ready
            maxI=terminateSelVarHighCost();
            . . . local min tabu list, reset code . . .
        }
        // Finish activities
        for(id in computeInst)
            computeInst(id).activityToDo = FINISH;
        startBarrier.wait();
        doneBarrier.wait();
```

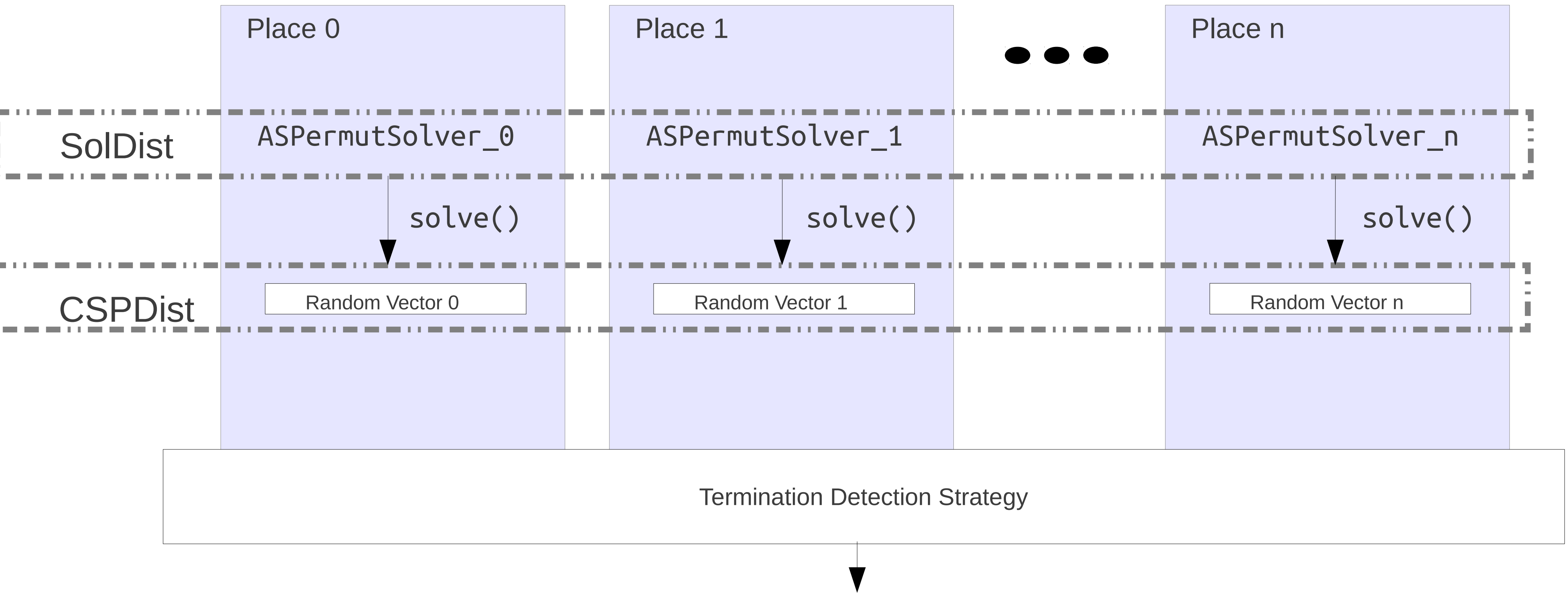
Main Loop

Second approach to functional parallelism

```
public class ASSolverFP1 extends ASPermutSolver{
    val computeInst : Array[ComputePlace];
    var startBarrier : ActivityBarrier;
    var doneBarrier : ActivityBarrier;
    public def solve(csp : CSPModel):Int{
        for(var th : Int = 1; th <= THNUM ; th++)
            computeInst(th) = new ComputePlace(th , csp);
        for(id in computeInst)
            async computeInst(id).run();
        while(total cost!=0){
            . . . restart code . . .
            for(id in computeInst)
                computeInst(id).activityToDo = SELECVARHIGHCOST;
            startBarrier.wait(); // send start signal
            // activities working...
            doneBarrier.wait(); // work ready
            maxI=terminateSelVarHighCost();
            . . . local min tabu list, reset code . . .
        }
        // Finish activities
        for(id in computeInst)
            computeInst(id).activityToDo = FINISH;
        startBarrier.wait();
        doneBarrier.wait();
```

Main Loop

Adaptive Search – X10
Data Parallelism



The Adaptive Search Method X10 Implementation

```
public class ASSolverRW{
  val solDist : DistArray [ASPermutSolver];
  val cspDist : DistArray [CSPModel];
  public def solve(){
    val random = new Random();
    finish for( p in Place.places() ){
      val seed = random.nextLong();
      at(p) async {
        CspDist( here.id ) = new CSPModel(seed);
        SolDist( here.id ) = new ASPermutSolver(seed);
        cost = solDist(here.id).solve(cspDist(here.id));
        if (cost==0) {
          for (k in Place.places())
            if (here.id != k.id)
              at(k) async {
                solDist(here.id).kill = true;
              }
        }
      }
    }
   return cost;
```

The Adaptive Search Method X10 Implementation

```
public class ASSolverRW{
  val solDist : DistArray [ASPermutSolver];
  val cspDist : DistArray [CSPModel];
  public def solve(){
    val random = new Random();
    finish for( p in Place.places() ){
      val seed = random.nextLong();
      at(p) async {
        CspDist( here.id ) = new CSPModel(seed);
        SolDist( here.id ) = new ASPermutSolver(seed);
        cost = solDist(here.id).solve(cspDist(here.id));
        if (cost==0) {
          for (k in Place.places())
            if (here.id != k.id)
              at(k) async {
                solDist(here.id).kill = true;
              }
        }
      }
    }
   return cost;
```

IRW

The Adaptive Search Method X10 Implementation

```
public class ASSolverRW{
  val solDist : DistArray [ASPermutSolver];
  val cspDist : DistArray [CSPModel];
  public def solve(){
    val random = new Random();
    finish for( p in Place.places() ){
      val seed = random.nextLong();
      at(p) async {
        CspDist( here.id ) = new CSPModel(seed);
        SolDist( here.id ) = new ASPermutSolver(seed);
        cost = solDist(here.id).solve(cspDist(here.id));
        if (cost==0) {
          for (k in Place.places())
            if (here.id != k.id)
              at(k) async {
                solDist(here.id).kill = true;
              }
        }
      }
    }
   return cost;
```

IRW

The Adaptive Search Method X10 Implementation

```
public class ASSolverRW{
  val solDist : DistArray [ASPermutSolver];
  val cspDist : DistArray [CSPModel];
  public def solve(){
    val random = new Random();
    finish for( p in Place.places() ){
      val seed = random.nextLong();
      at(p) async {
        CspDist( here.id ) = new CSPModel(seed);
        SolDist( here.id ) = new ASPermutSolver(seed);
        cost = solDist(here.id).solve(cspDist(here.id));
        if (cost==0) {
          for (k in Place.places())
            if (here.id != k.id)
              at(k) async {
                solDist(here.id).kill = true;
              }
        }
      }
    }
   return cost;
```

IRW

The Adaptive Search Method X10 Implementation

```
public class ASSolverRW{
  val solDist : DistArray [ASPermutSolver];
  val cspDist : DistArray [CSPModel];
  public def solve(){
    val random = new Random();
    finish for( p in Place.places() ){
      val seed = random.nextLong();
      at(p) async {
        CspDist( here.id ) = new CSPModel(seed);
        SolDist( here.id ) = new ASPermutSolver(seed);
        cost = solDist(here.id).solve(cspDist(here.id));
        if (cost==0) {
          for (k in Place.places())
            if (here.id != k.id)
              at(k) async {
                solDist(here.id).kill = true;
              }
        }
      }
    }
   return cost;
```

IRW

The Adaptive Search Method X10 Implementation

```
public class ASSolverRW{
  val solDist : DistArray [ASPermutSolver];
  val cspDist : DistArray [CSPModel];
  public def solve(){
    val random = new Random();
    finish for( p in Place.places() ){
      val seed = random.nextLong();
      at(p) async {
        CspDist( here.id ) = new CSPModel(seed);
        SolDist( here.id ) = new ASPermutSolver(seed);
        cost = solDist(here.id).solve(cspDist(here.id));
        if (cost==0) {
          for (k in Place.places())
            if (here.id != k.id)
              at(k) async {
                solDist(here.id).kill = true;
              }
        }
      }
    }
   return cost;
```

TD

The Adaptive Search Method X10 Implementation

```
public class ASSolverRW{
  val solDist : DistArray [ASPermutSolver];
  val cspDist : DistArray [CSPModel];
  public def solve(){
    val random = new Random();
    finish for( p in Place.places() ){
      val seed = random.nextLong();
      at(p) async {
        CspDist( here.id ) = new CSPModel(seed);
        SolDist( here.id ) = new ASPermutSolver(seed);
        cost = solDist(here.id).solve(cspDist(here.id));
        if (cost==0) {
          for (k in Place.places())
            if (here.id != k.id)
              at(k) async {
                solDist(here.id).kill = true;
              }
        }
      }
    }
   return cost;
```

TD

The Adaptive Search Method X10 Implementation

```
public class ASSolverRW{
  val solDist : DistArray [ASPermutSolver];
  val cspDist : DistArray [CSPModel];
  public def solve(){
    val random = new Random();
    finish for( p in Place.places() ){
      val seed = random.nextLong();
      at(p) async {
        CspDist( here.id ) = new CSPModel(seed);
        SolDist( here.id ) = new ASPermutSolver(seed);
        cost = solDist(here.id).solve(cspDist(here.id));
        if (cost==0) {
          for (k in Place.places())
            if (here.id != k.id)
              at(k) async {
                solDist(here.id).kill = true;
              }
        }
      }
    }
   return cost;
```

TD

The Adaptive Search Method X10 Implementation

```
public class ASSolverRW{
  val solDist : DistArray [ASPermutSolver];
  val cspDist : DistArray [CSPModel];
  public def solve(){
    val random = new Random();
    finish for( p in Place.places() ){
      val seed = random.nextLong();
      at(p) async {
        CspDist( here.id ) = new CSPModel(seed);
        SolDist( here.id ) = new ASPermutSolver(seed);
        cost = solDist(here.id).solve(cspDist(here.id));
        if (cost==0) {
          for (k in Place.places())
            if (here.id != k.id)
              at(k) async {
                solDist(here.id).kill = true;
              }
        }
      }
    }
   return cost;
```

Agenda

- Context
- X10 programming language
- Adaptive Search
  - Parallel Implementations
- Experimentation Results
- Conclusion and Future Work

Experimentation

- Benchmark Set:
  - Magic Square Problem (**MSP**)
  - Number Partitioning Problem (**NPP**)
  - All-Interval Problem (**AIP**)
  - Costas Array Problem (**CAP**)
- Hardware Platform:
  - Non-uniform memory access (NUMA) computers
    - 2 Intel Xeon W5580 CPUs each one with 4 hyper-threaded cores running at 3.2GHz
    - 4 16-core AMD Opteron 6272 CPUs running at 2.1GHz

Results – Functional Parallelism

- First Approach

Problem instance	time (s) seq.	speed-up with k places 2	4	8	time (s) 8 places
MSP-100	11.98	0.86	0.95	0.77	15.49
MSP-120	24.17	1.04	0.97	0.98	24.65
CAP-17	1.56	0.43	0.28	0.24	6.53
CAP-18	12.84	0.51	0.45	0.22	57.16

- Second Approach

Problem instance	time (s) seq.	speed-up with k places 2	4	8	time (s) 8 places
MSP-100	11.98	1.15	0.80	0.86	13.87
MSP-120	24.17	1.23	0.94	0.63	38.34
CAP-17	1.56	0.56	0.30	0.25	6.35
CAP-18	12.84	0.74	0.39	0.27	46.84

Results – Functional Parallelism

- First Approach

Problem instance	time (s) seq.	speed-up with k places 2	4	8	time (s) 8 places
MSP-100	11.98	0.86	0.95	0.77	15.49
MSP-120	24.17	1.04	0.97	0.98	24.65
CAP-17	1.56	0.43	0.28	0.24	6.53
CAP-18	12.84	0.51	0.45	0.22	57.16

- Second Approach

Problem instance	time (s) seq.	speed-up with k places 2	4	8	time (s) 8 places
MSP-100	11.98	1.15	0.80	0.86	13.87
MSP-120	24.17	1.23	0.94	0.63	38.34
CAP-17	1.56	0.56	0.30	0.25	6.35
CAP-18	12.84	0.74	0.39	0.27	46.84

Results – Functional Parallelism

- First Approach

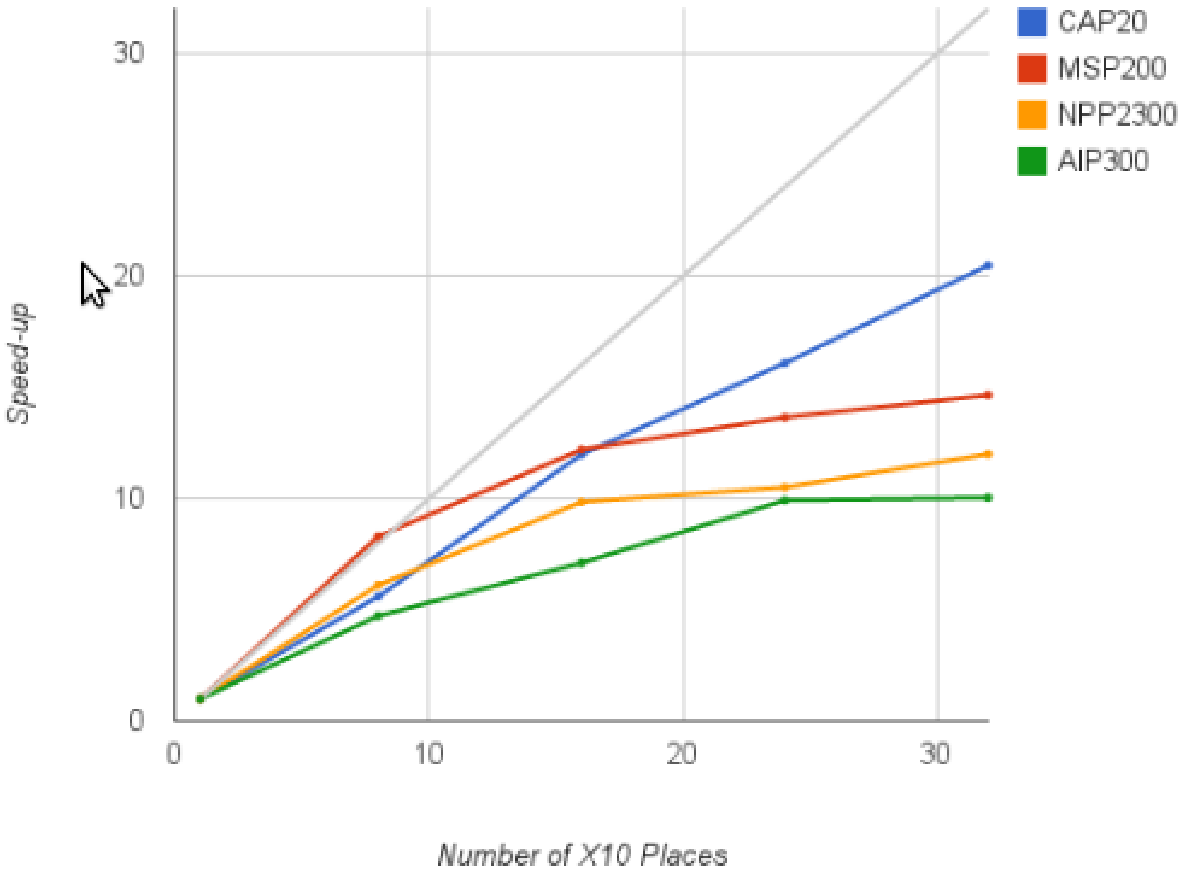
Problem instance	time (s) seq.	speed-up with k places			time (s) 8 places
		2	4	8	
MSP-100	11.98	0.86	0.95	0.77	15.49
MSP-120	24.17	$\boxed{1.04}$	0.97	0.98	24.65
CAP-17	1.56	0.43	0.28	0.24	6.53
CAP-18	12.84	0.51	0.45	0.22	57.16

- Second Approach

Problem instance	time (s) seq.	speed-up with k places			time (s) 8 places
		2	4	8	
MSP-100	11.98	1.15	0.80	0.86	13.87
MSP-120	24.17	1.23	0.94	0.63	38.34
CAP-17	1.56	0.56	0.30	0.25	6.35
CAP-18	12.84	0.74	0.39	0.27	46.84

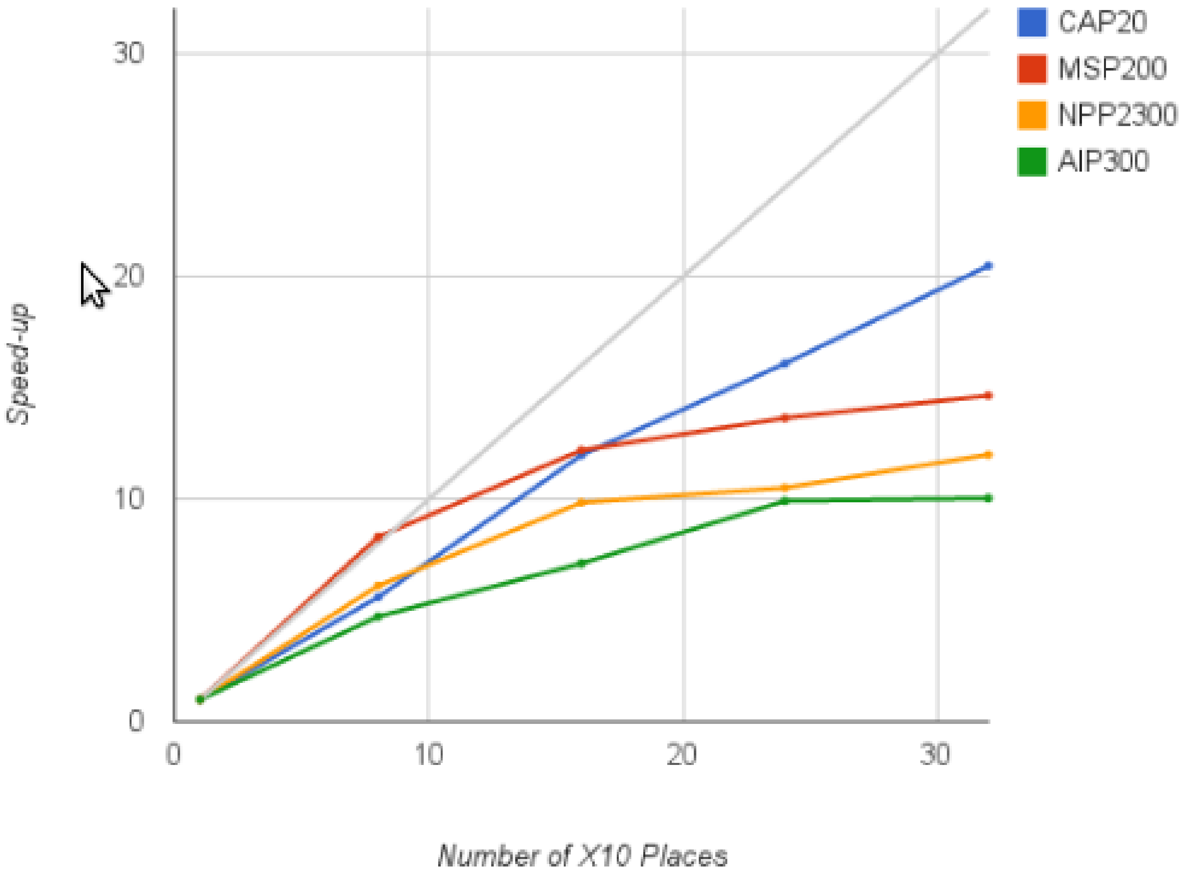
Results - Data Parallelism

Problem instance	time (s) seq.	speed-up with k places				time (s) 32 places
		8	16	24	32	
AIP-300	56.7	4.7	7.1	9.9	10.0	5.6
NPP-2300	6.6	6.1	9.8	10.5	12.0	0.5
MSP-200	365	8.3	12.2	13.6	14.6	24.9
CAP-20	731	5.6	12.0	16.1	20.5	35.7



Results - Data Parallelism

Problem instance	time (s) seq.	speed-up with k places 8	16	24	32	time (s) 32 places
AIP-300	56.7	4.7	7.1	9.9	10.0	5.6
NPP-2300	6.6	6.1	9.8	10.5	12.0	0.5
MSP-200	365	8.3	12.2	13.6	14.6	24.9
CAP-20	731	5.6	12.0	16.1	20.5	35.7



Agenda

- Context
- X10 programming language
- Adaptive Search
  - Parallel Implementations
- Experimentation Results
- Conclusion and Future Work

Conclusion

- Parallel X10 implementation Adaptive Search:
  - So far and under the current test conditions, **Functional Parallelism yields no speed-up**.
  - **Good level of performance** for the X10 data-parallelism implementation.
- **Linear (or close) speed-ups**.

Conclusion

- X10 is a suitable platform to exploit parallelism in different ways
  - Thanks to X10 we can experiment various strategies:
    - Single shared memory inter-process parallelism
    - Distributed memory programming model.
- **X10 implicit communication mechanisms** (abstractions)
  - The distributed arrays and the termination detection system in our data parallel implementation.

Future Work

- **Cooperative Local Search** parallel solver using data parallelism.
  - Taking advantage of all communications tools available in **X10**.
- Test the behavior of a cooperative implementation, under different **HPC architectures**.
  - **Many-core Architectures: Xeon PHI, GPGPU.**
  - **Grid computing platforms like Grid5000.**
- Compare with other programming tools.

Thank you!!!

Questions?

Contact: Danny.Munera@malix.univ-paris1.fr
Université Paris 1
Panheon-Sorbone

How to improve solving performance?

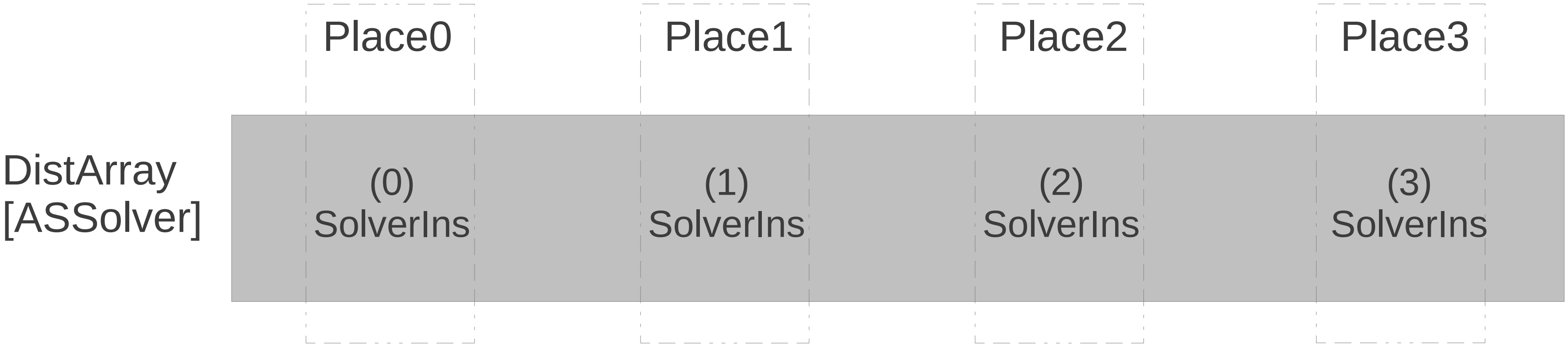
- More computational resourses:
  - **PARALLELISM**
- Great diversity:
  - Multi-core Many-core Processors
  - Computer Cluster
  - Grid computing
  - GPGPUs

X10 in a Nutshel

- **Synchronization:**
  - **Finish:** to wait the termination of a set of activities.
  - **Atomic:** ensures an exclusive access to a critical portion of code.
  - **Clocks:** standard way to ensure the synchronization between activities or places.
  - ...
- Distibuted Arrays, GlobalRefs, etc...

Distributed Arrays

- Arrays provide indexed access to data at a single Place, via *Points* - indices of any dimensionality.
- DistArrays is similar, but spreads the data across multiple Places.

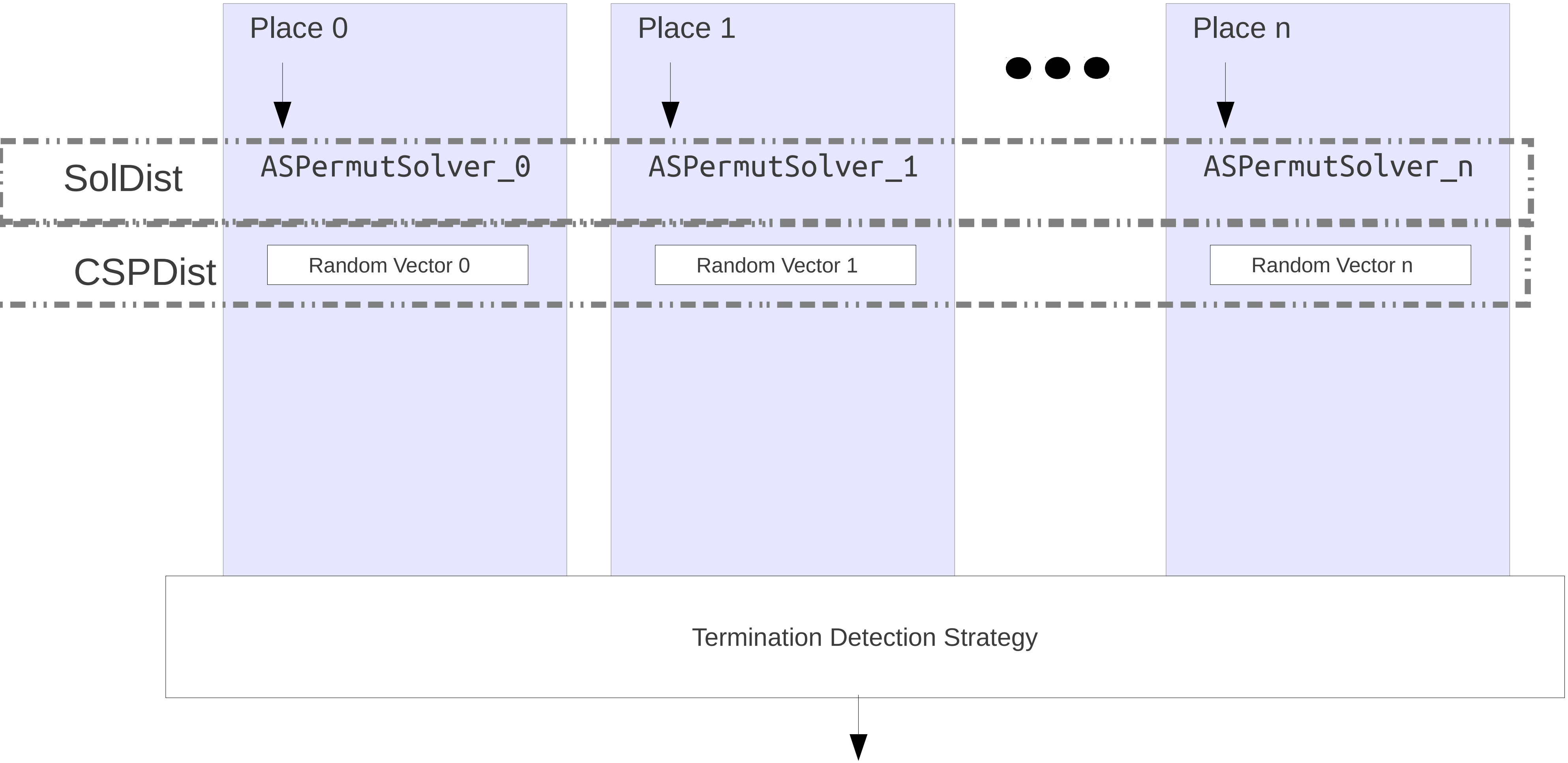


The Adaptive Search Method X10 Implementation

- Functional Parallelism
  - Inner Loop
  - Fine-grained Parallelism
    - Using Activities

  - High activities managment overhead
  - No good results

RW graph



Place 0
Place 1
Place n
SolDist
ASPermutSolver_0
ASPermutSolver_1
ASPermutSolver_n
CSPDist
Random Vector 0
Random Vector 1
Random Vector n
Termination Detection Strategy

The Adaptive Search Method X10 Implementation

- Functional Parallelism: No good results
- Data Parallelism
  - The **search space is divided** using different random start points (i.e configurations)
  - Independent Random Walks
    - When one instance reaches a solution, a **termination detection communication strategy** is used to finalize the remaining running instances.

Banchmark description

- The speed-up increases **almost linearly** with the number of places used in the X10 program
- For some problems the speed-up seems to **increase with the size** of the problem.
- The results are as good as reported in the literature when using other IPC frameworks such as MPI.

Banchmark description

- Magic Square Problem (MSP)
- The Magic Square Problem (prob019 in CSPLib)
- Consists of placing on a N × N square all the numbers in {1, 2, . . . , N^2 } such as the sum of the numbers in all rows, columns and the two diagonal are the same.
- N^2 variables with initial domains {1, 2, . . . , N^2 } together with linear equation constraints and a global all-different constraint stating that all variables should have a different value.
- The constant value that should be the sum of all rows, columns and the two diagonals can be easily computed to be N (N 2 + 1)/2.

Banchmark description

- All-Interval Problem (AIP)
- The All-Interval Problem (prob007 in CSPLib)
- Consists of composing a sequence of N notes such that all are different and tonal intervals between consecutive notes are also distinct. This problem is equivalent to finding a permutation of the N first integers such that the absolute difference between two consecutive pairs of numbers are all different.
- Find a permutation $(X_1, \ldots, X_N)$ of $(0, \ldots, N-1)$ such that the list $(abs(X_1 - X_2), abs(X_2 - X_3), \ldots, abs(X_{N-1} - X_N))$ is a permutation of $(1, \ldots, N-1)$.
- A possible solution for N = 8 is (3, 6, 0, 7, 2, 4, 5, 1) because all consecutive distances are different.

Banchmark description

- Number Partitioning Problem (NPP)
- The Number Partitioning Problem (prob049 in CSPLib)
- Consists of finding a partition of numbers {1, . . . , N } into two groups A and B of the same cardinality such that the sum of numbers in A is equal to the sum of numbers in B and the sum of squares of numbers in A is equal to the sum of squares of numbers in B.
- A solution for N = 8 is A = {1, 4, 6, 7} and B = {2, 3, 5, 8}.

Banchmark description

- Costas Array Problem (CAP)
- The Costas Array Problem consists of filling an $N \times N$ grid with N marks such that there is exactly one mark per row and per column and the $N (N - 1)/2$ vectors joining the marks are all different. It is convenient to see the Costas Array Problem as a permutation problem by considering an array of N variables $(X1 , \ldots , Xn)$ which forms a permutation of $\{1, 2, \ldots , N \}$ subject to some all-different constraints
- This problem has many practical applications and currently it has a whole community active working around it (www.costasarrays.org/).

Results

- The speed-up increases **almost linearly** with the number of places used in the X10 program
- For some problems the speed-up seems to **increase with the size** of the problem.
- The results are as good as reported in the literature when using other IPC frameworks such as MPI.