AImotion Bavaria

Evaluation of Parallelization Frameworks for the Linear Assignment Problem

PARS Workshop 2025

17.09.2025

Jakob Stadelmann, Andreas Schweiger, Richard Membarth

dfki ai

AIRBUS

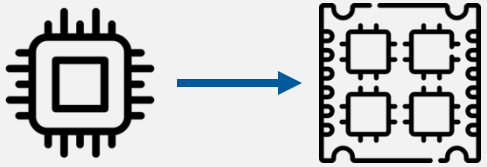
Technische Hochschule Ingolstadt

Work supported by BMBF as part of the MANNHEIM-AutoDevSafeOps and by StMWi as part of the DFOP-HD project.

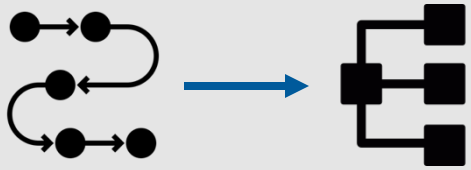
Agenda

- Problem Statement
- Parallelization Mechanisms
- Linear Assignment Problem
- Implementation
- Evaluation

Problem Statement



Modern embedded systems are increasingly transitioning from single-core to multi-core architectures to meet growing performance demands.



Legacy software must be restructured into independent, parallelizable components that can be executed concurrently on multiple cores.

Parallelization of such software is a complex, error-prone, and resource-intensive process.

Motivation:

There is a clear need for efficient parallelization approaches for embedded environments, which reduce the development effort while preserving the functionality and improving performance.

Parallelization Mechanisms

POSIX Threads

- Low-level threading API for Unix-like systems
- Link with the pthread library
- Explicit thread control
- Synchronization via primitives

Pros: fine-grained control, high potential efficiency on custom workloads.

Cons: manual synchronization, explicit load balancing, higher risk of race conditions and bugs.

OpenMP API

- Pragmas/directives-based parallelism
- Annotate loops or task regions
- Supports common patterns
- Synchronization via directives

Pros: easy to adopt, minimal code changes, portable.

Cons: less low-level control, possible overhead, still need to avoid races and tune parallelism.

Parallelization Mechanisms

emmtrix Parallel Studio (ePS)

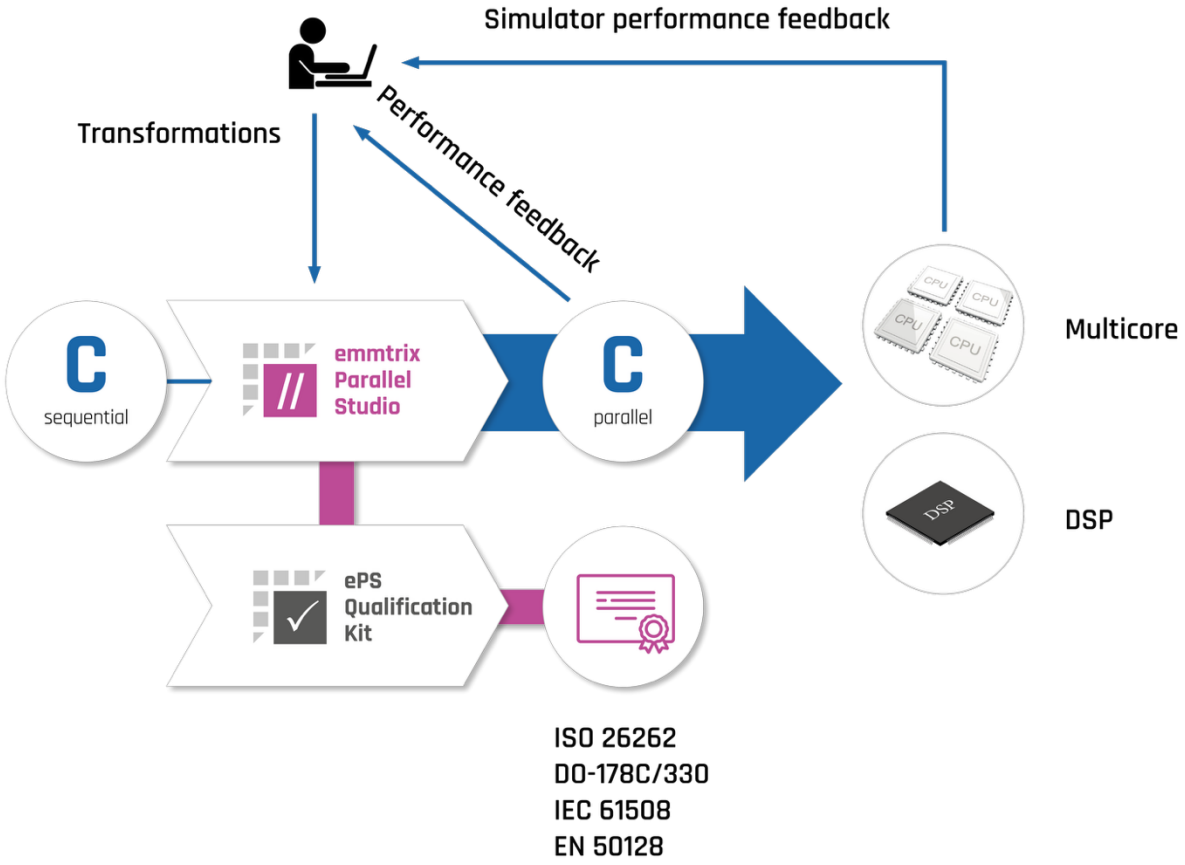
Source-to-source transformation tool.

Features:

- Automated, guided transformation reduces manual parallelization effort
- Correct-by-design approach
- Fine-grained transformations available via GUI

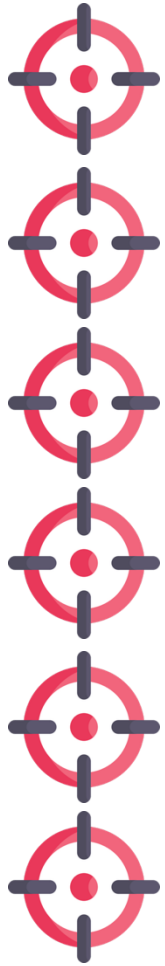
Limitations:

- Requires **MISRA C** input: forbids dynamic allocation, recursion, flexible pointer patterns, etc.

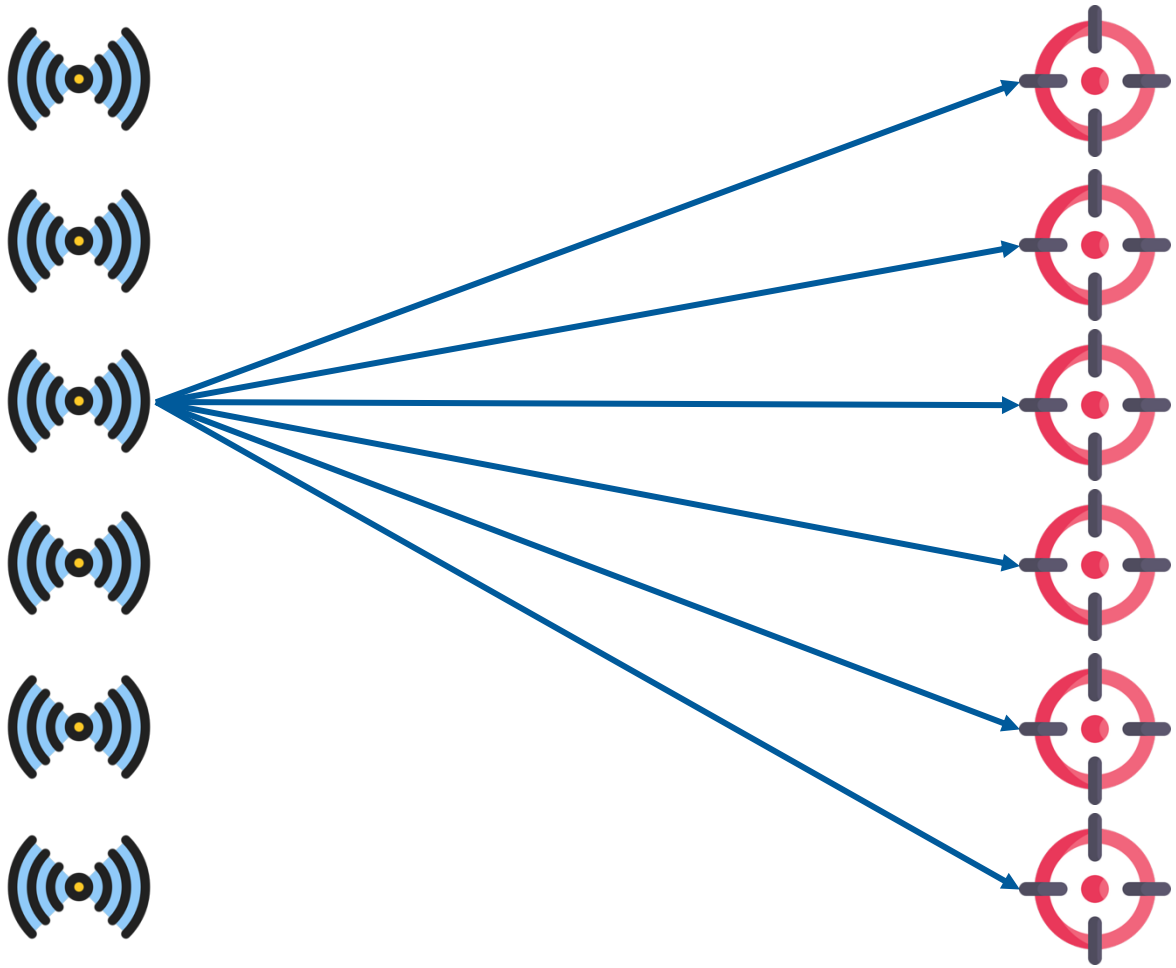


[1] emmtrix Technologies GmbH

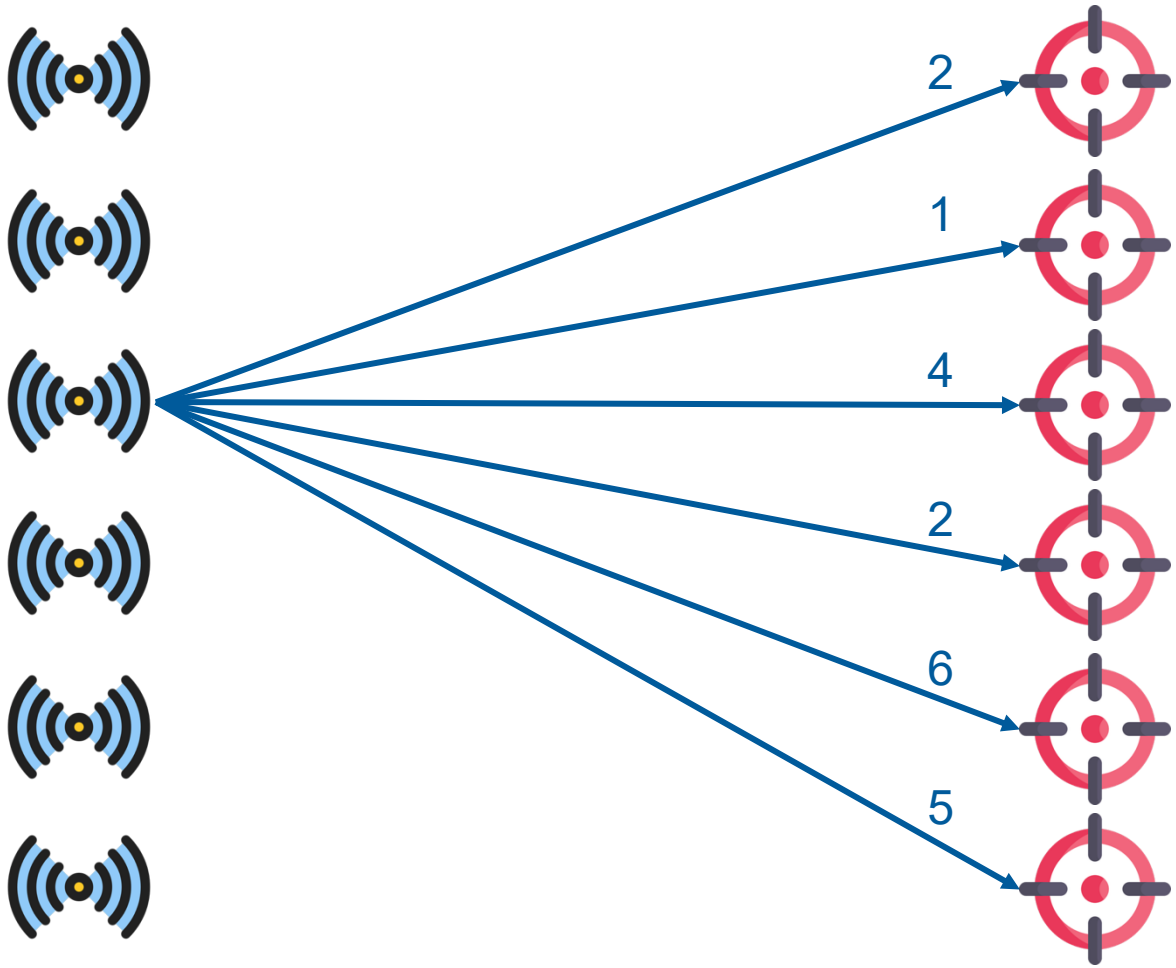
Linear Assignment Problem (LAP)



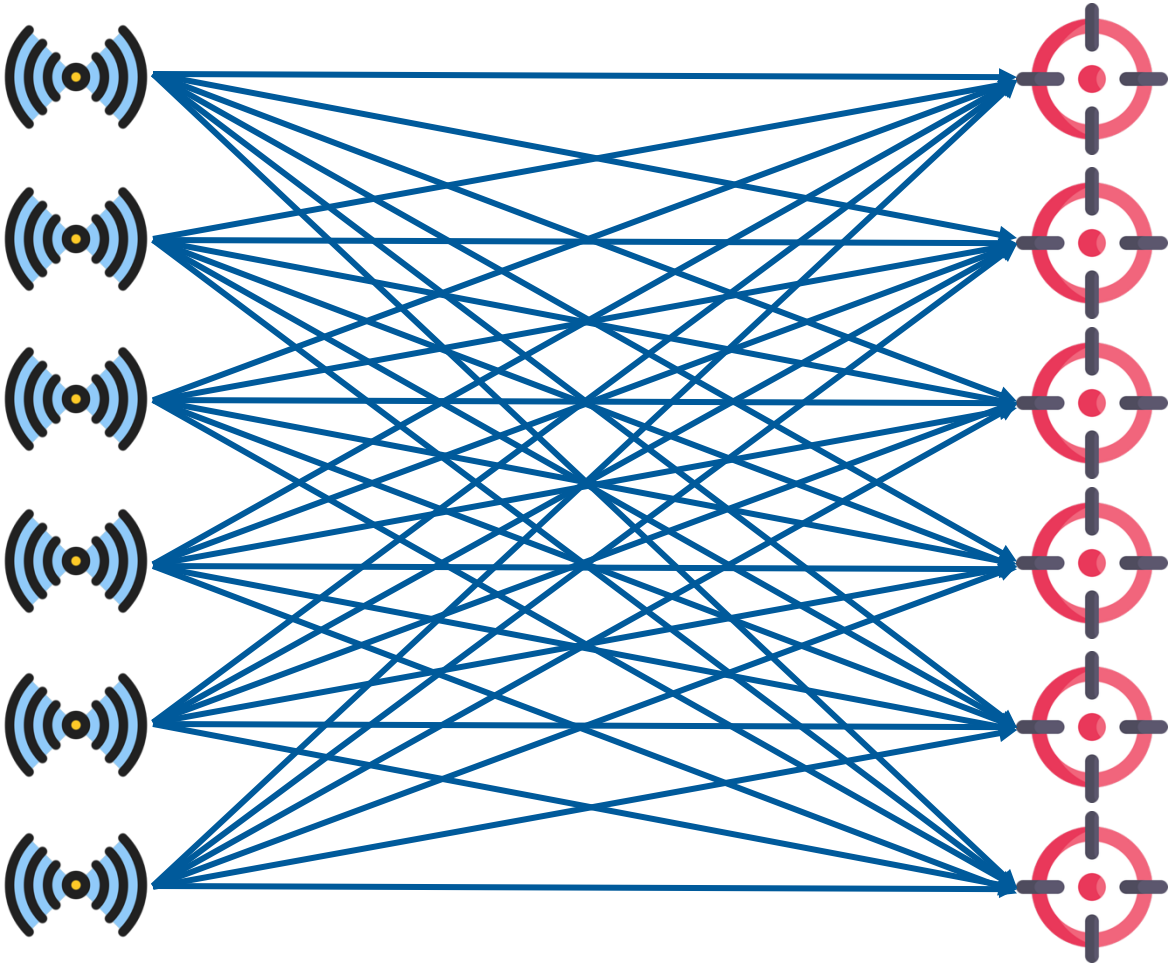
Linear Assignment Problem (LAP)



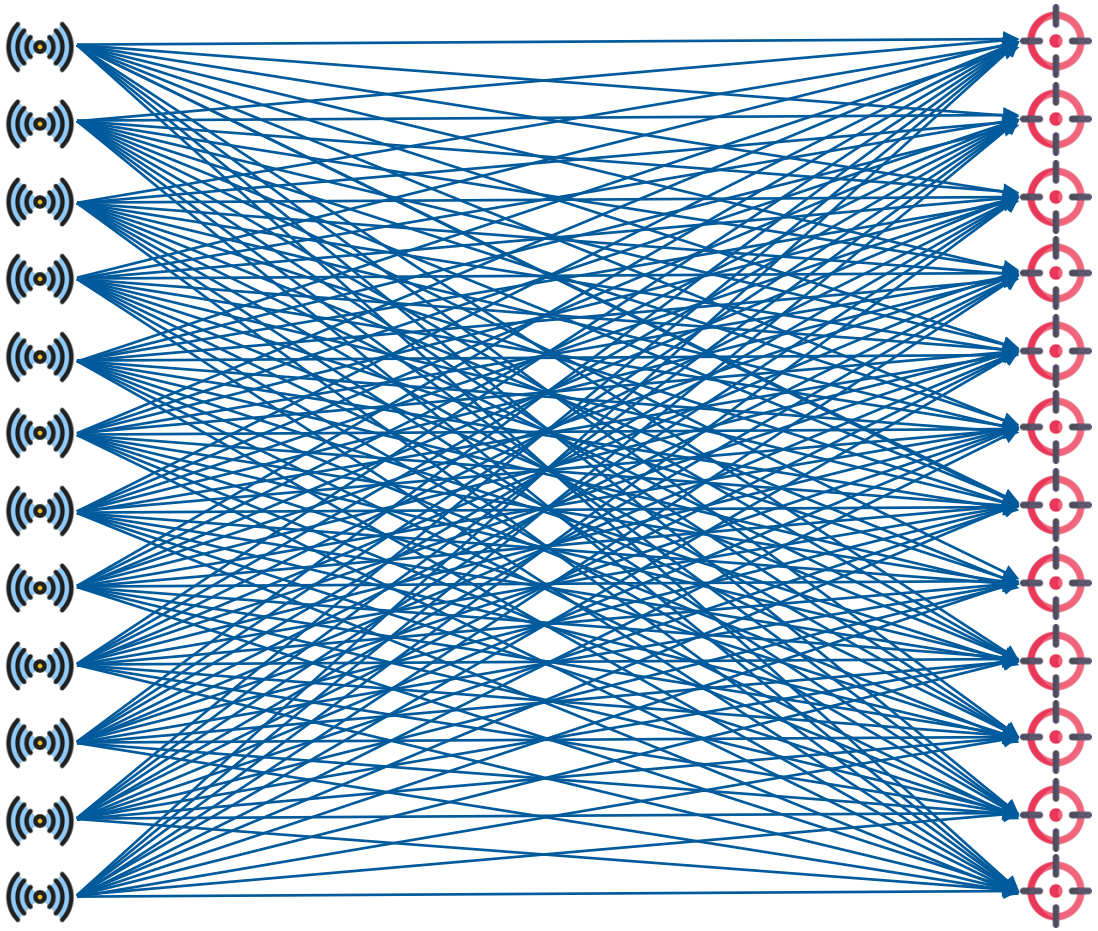
Linear Assignment Problem (LAP)



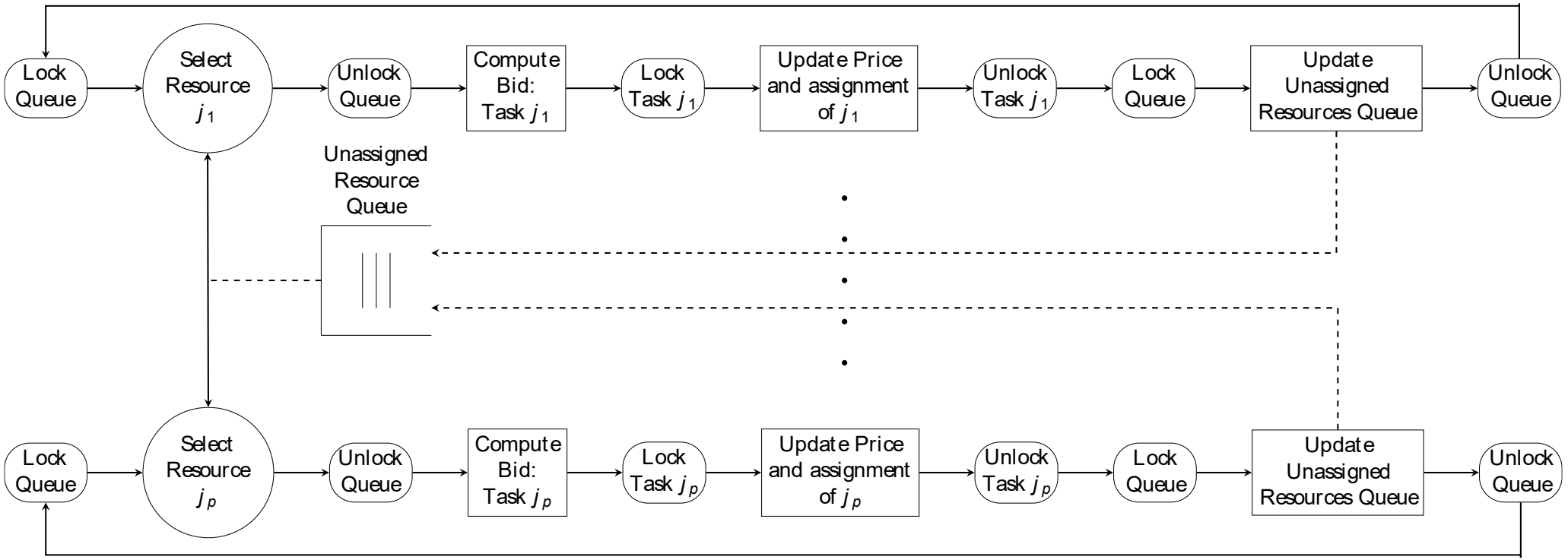
Linear Assignment Problem (LAP)



Linear Assignment Problem (LAP)

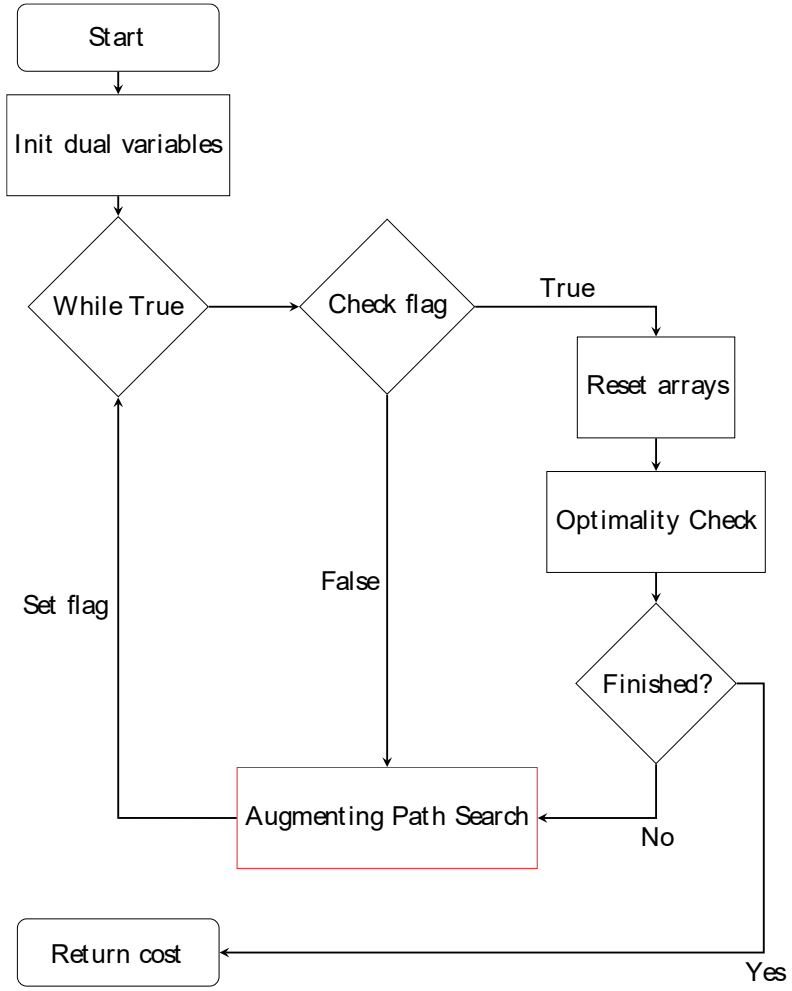


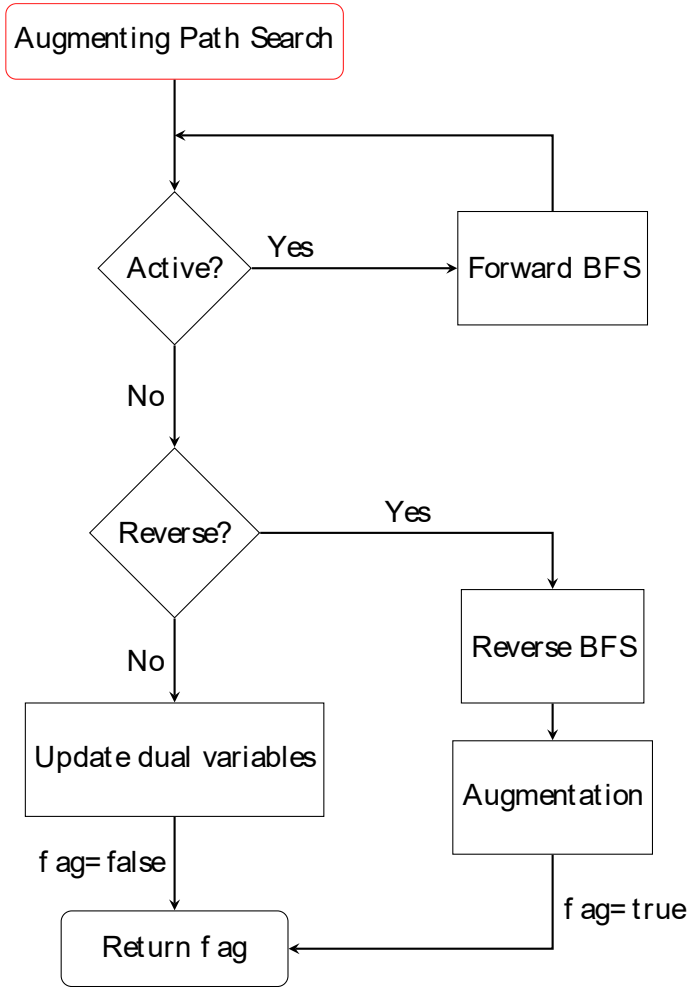
Auction Implementation



[2] Dimitri P. Bertsekas and David A. Castañon

Hungarian Implementation





[3] Ketan Date and Rakesh Nagi

Serial Implementation - Example

```cpp
void init_potentials(vector<vector<int>> &cost, int n, vector<double>& dual_row, vector<double>& dual_col){
    // Init dual row
    for (int i = 0; i < n; i++){
        double temp = inf;
        // Find minimum value of row i
        for (int j = 0; j < n; j++){
            temp = min(temp, double(cost[i][j]));
        }
        dual_row[i] = temp;
    }

    // Init dual col
    for (int j = 0; j < n; j++){
        double temp = inf;
        // Find minimum value of col j
        for (int i = 0; i < n; i++){
            temp = min(temp, cost[i][j] - dual_row[i]);
        }
        dual_col[j] = temp;
    }
}
```

First phase:

- For each row in the cost matrix, find minimum value across all columns

Second phase:

- For each column, calculate reduced costs by subtracting the corresponding row dual

Caution:

- Second phase depends on the completed dual_row values from first phase

OpenMP Implementation - Example

```
void init_potentials_parallel(vector<vector<int>> &cost, int n, vector<double>& dual_row, vector<double>& dual_col){
    // Init dual row
    #pragma omp parallel for num_threads(NUM_THREADS)
    for (int i = 0; i < n; i++){
        double temp = inf;
        // Find minimum value of row i
        for (int j = 0; j < n; j++){
            temp = min(temp, double(cost[i][j]));
        }
        dual_row[i] = temp;
    }

    // Init dual col
    #pragma omp parallel for num_threads(NUM_THREADS)
    for (int j = 0; j < n; j++){
        double temp = inf;
        // Find minimum value of col j
        for (int i = 0; i < n; i++){
            temp = min(temp, cost[i][j] - dual_row[i]);
        }
        dual_col[j] = temp;
    }
}
```

- Uses *#pragma omp parallel for num_threads(NUM_THREADS)* to distribute work across threads
- OpenMP ensures automatic synchronization between phases

pthreads Implementation - Example

```cpp
void init_potentials_parallel(){
    // Schedule initialization of dual row
    for (int i=0; i < NUM_THREADS; i++){
        thpool_.schedule([i]{
            init_dual_row(i);
        }, i);
    }
    for (int i=0; i < NUM_THREADS; i++){
        thpool_.wait(i);
    }

    // Schedule initialization of dual col
    for (int i=0; i < NUM_THREADS; i++){
        thpool_.schedule([i]{
            init_dual_col(i);
        }, i);
    }
    for (int i=0; i < NUM_THREADS; i++){
        thpool_.wait(i);
    }

}
```

```cpp
void init_dual_row(int thread_id){
    const int chunk = div_ceil(n, NUM_THREADS); // chunk size
    const int lower = thread_id * chunk; // lower task id
    const int upper = min(lower + chunk, n); // upper task id

    // Init dual row
    for (int i = lower; i < upper; i++){
        double temp = inf;
        // Find minimum value of row i
        for (int j = 0; j < n; j++){
            temp = min(temp, double(cost[i][j]));
        }
        dual_row[i] = temp;
    }
}

void init_dual_col(int thread_id){
    const int chunk = div_ceil(n, NUM_THREADS); // chunk size
    const int lower = thread_id * chunk; // lower task id
    const int upper = min(lower + chunk, n); // upper task id

    // Init dual col
    for (int j = lower; j < upper; j++){
        double temp = inf;
        // Find minimum value of col j
        for (int i = 0; i < n; i++){
            temp = min(temp, cost[i][j] - dual_row[i]);
        }
        dual_col[j] = temp;
    }
}
```

- Extract logic of first and second phase into functions
- Explicit synchronization between phases prevents race conditions
- Uses threadpool for efficient thread management
- Thread Pool Management:
  - thpool_.schedule() assigns lambda functions to specific threads
  - thpool_.wait() ensures all threads complete before proceeding
- Work Distribution Strategy:
  - Chunk-based partitioning
  - Each thread processes a contiguous range: [lower, upper)

ePS Implementation - Example

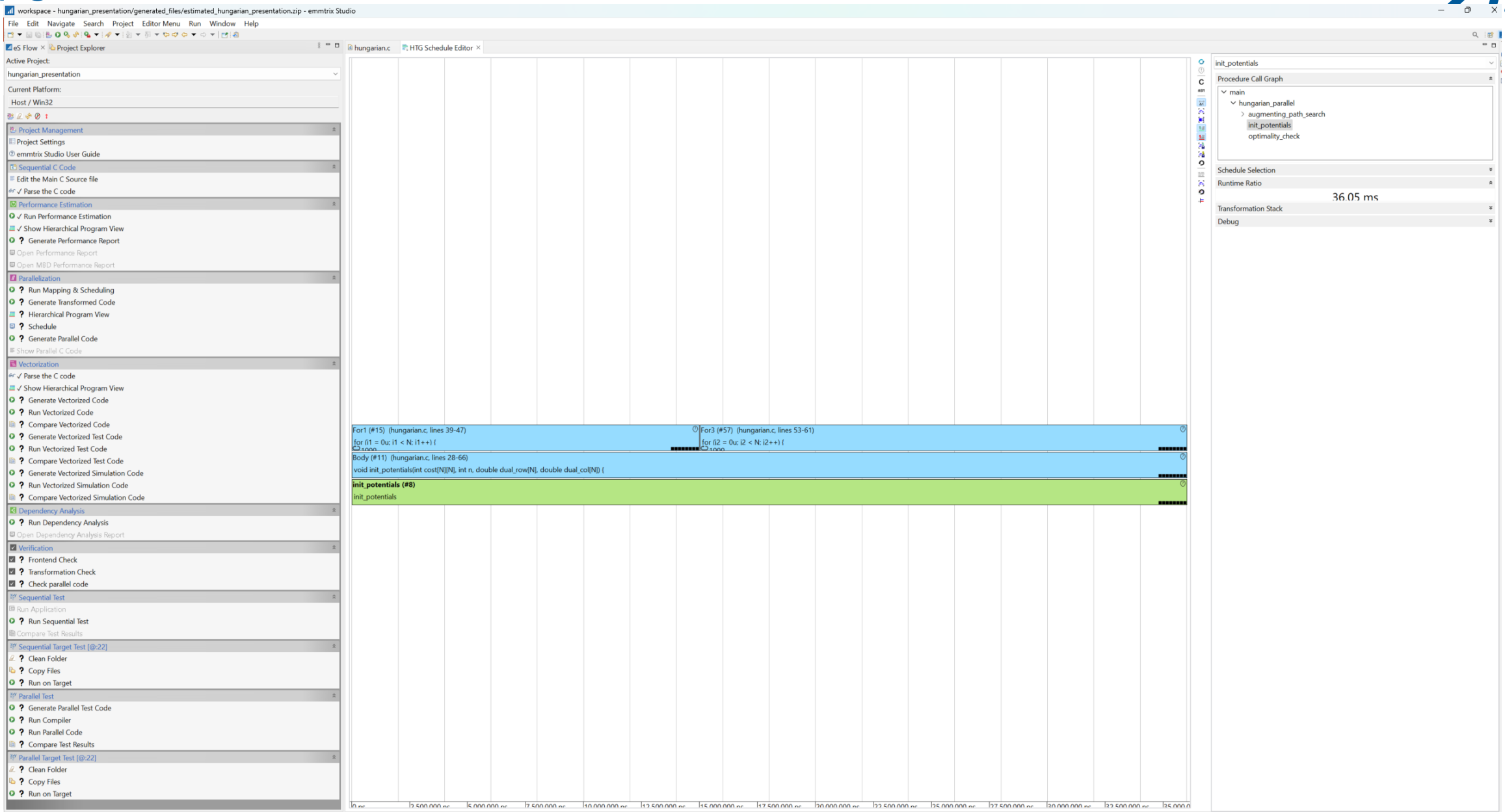
```
void init_potentials(int cost[N][N], int n, double dual_row[N], double dual_col[N]) {
    size_t i1;
    size_t i2;
    size_t i3;
    size_t i4;
    size_t j1;
    size_t j2;

    double d_row[N];
    double d_col[N];

    for (i1 = 0u; i1 < N; i1++) {
        double temp = inf;
        for (j1 = 0u; j1 < N; j1++) {
            if (temp > (double)cost[i1][j1]){
                temp = (double)cost[i1][j1];
            }
        }
        d_row[i1] = temp;
    }

    for (i3=0u; i3<N; i3++){
        dual_row[i3] = d_row[i3];
    }

    for (j2 = 0u; j2 < N; j2++) {
        double temp = inf;
        for (i2 = 0u; i2 < N; i2++) {
            if (temp > (cost[i2][j2] - dual_row[i2])){
                temp = cost[i2][j2] - dual_row[i2];
            }
        }
        d_col[j2] = temp;
    }

    for (i4=0u; i4<N; i4++){
        dual_col[i4] = d_col[i4];
    }
}
```

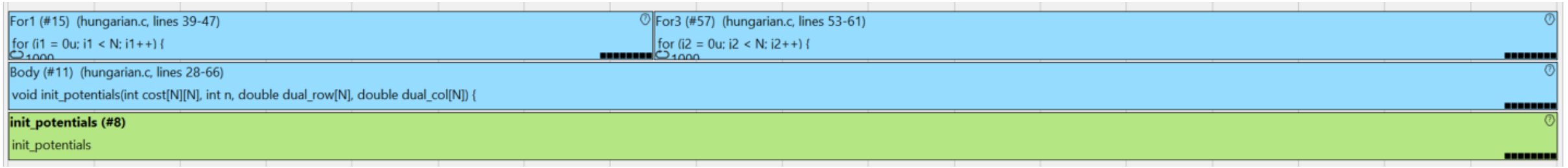
Serial code modifications:

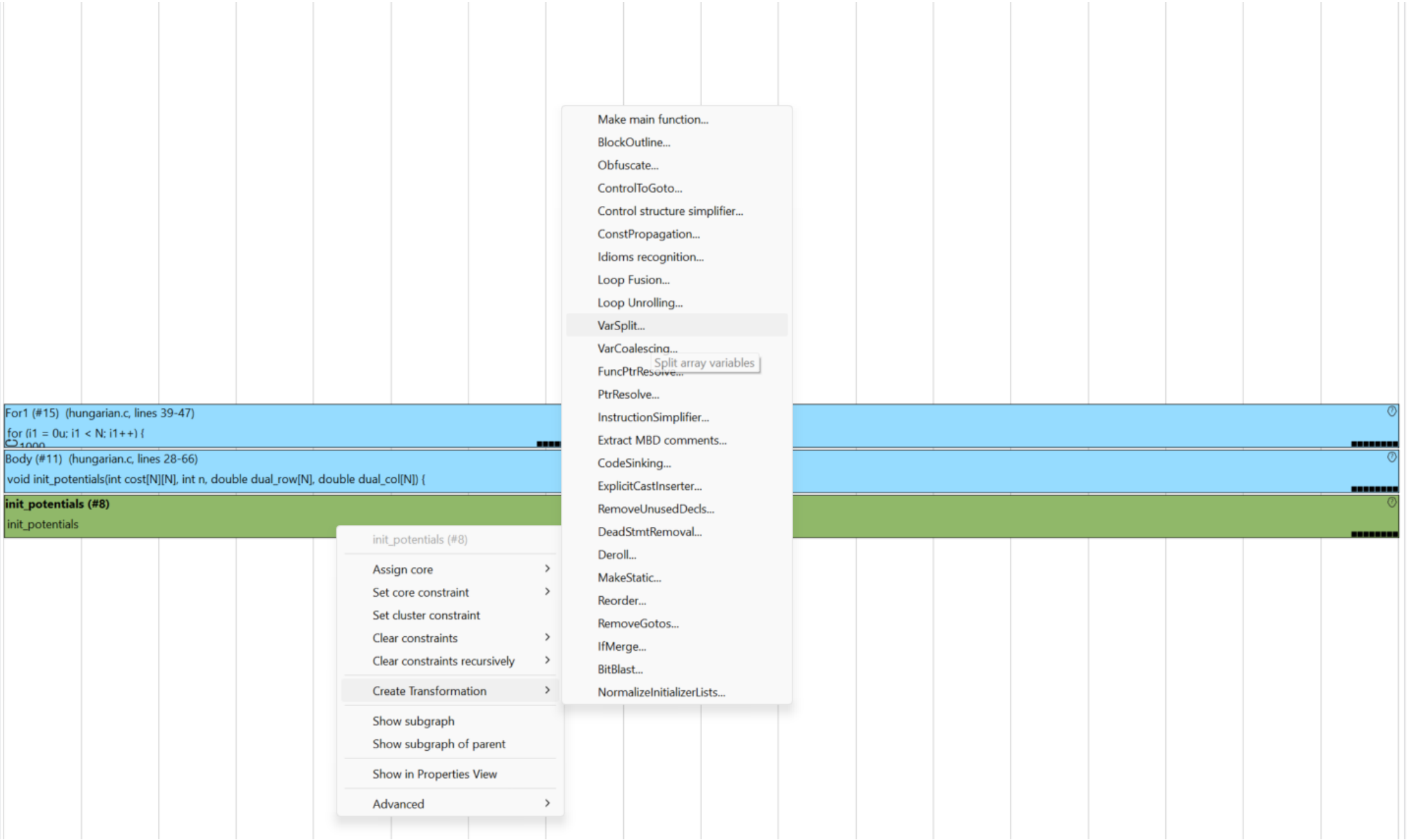
- Convert dynamic arrays to static arrays
- Preinitialize loop counters
- Create arrays to split and store partial results
- Introduce sequential logic to merge partial results

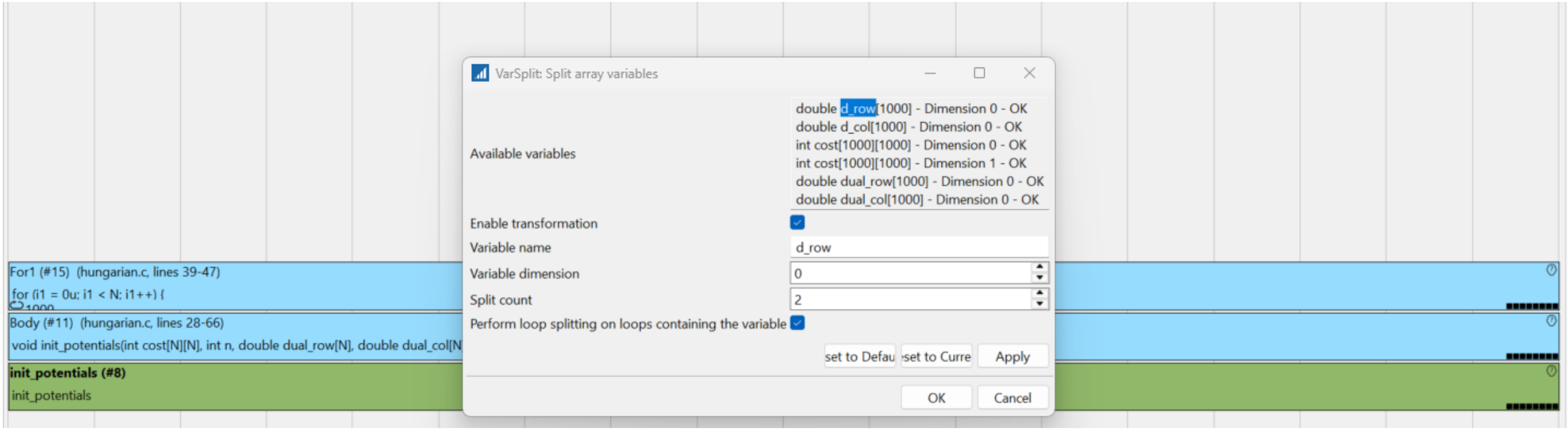
ePS transformations:

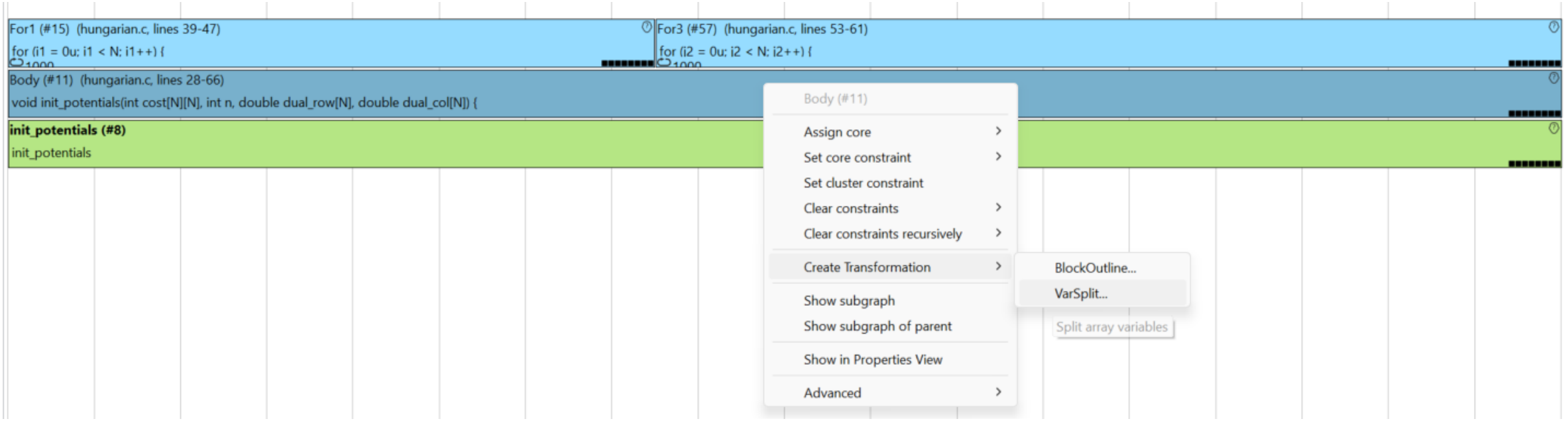
- Varsplit: d_row, d_col

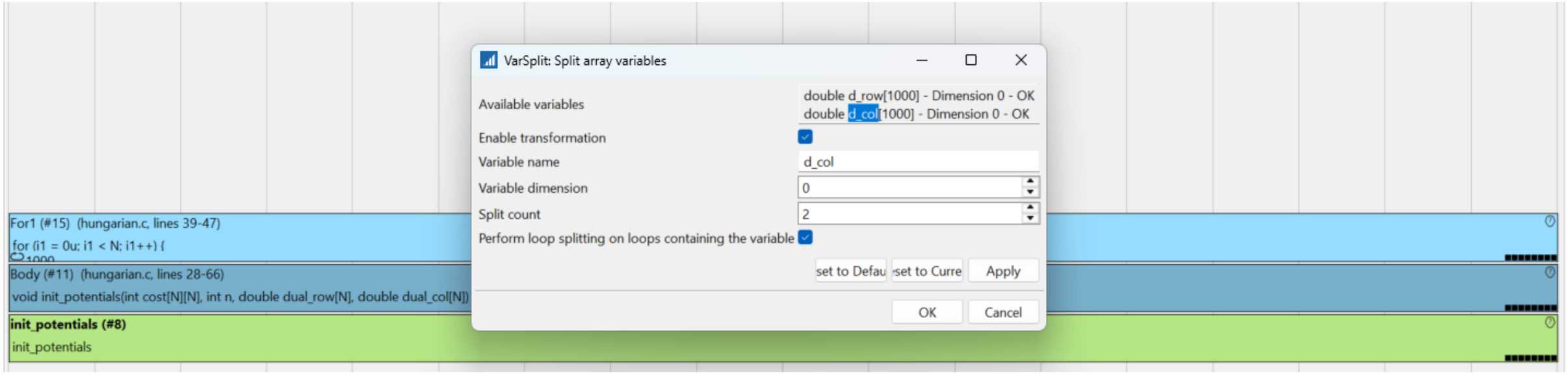

workspace - hungarian_presentation/generated_files/estimated_hungarian_presentation.zip - emmtrix Studio
File Edit Navigate Search Project Editor Menu Run Window Help
eS Flow
Project Explorer
Active Project:
hungarian_presentation
Current Platform:
Host / Win32
Project Management
Project Settings
emmtrix Studio User Guide
Sequential C Code
Edit the Main C Source file
Parse the C code
Performance Estimation
Run Performance Estimation
Show Hierarchical Program View
Generate Performance Report
Open Performance Report
Open MBD Performance Report
Parallelization
Run Mapping & Scheduling
Generate Transformed Code
Hierarchical Program View
Schedule
Generate Parallel Code
Show Parallel C Code
Vectorization
Parse the C code
Show Hierarchical Program View
Generate Vectorized Code
Run Vectorized Code
Compare Vectorized Code
Generate Vectorized Test Code
Run Vectorized Test Code
Compare Vectorized Test Code
Generate Vectorized Simulation Code
Run Vectorized Simulation Code
Compare Vectorized Simulation Code
Dependency Analysis
Run Dependency Analysis
Open Dependency Analysis Report
Verification
Frontend Check
Transformation Check
Check parallel code
Sequential Test
Run Application
Run Sequential Test
Compare Test Results
Sequential Target Test [@:22]
Clean Folder
Copy Files
Run on Target
Parallel Test
Generate Parallel Test Code
Run Compiler
Run Parallel Code
Compare Test Results
Parallel Target Test [@:22]
Clean Folder
Copy Files
Run on Target
hungarian.c
HTG Schedule Editor
For1 (#15) (hungarian.c, lines 39-47)
for (i1 = 0u; i1 < N; i1++) {
For3 (#57) (hungarian.c, lines 53-61)
for (i2 = 0u; i2 < N; i2++) {
Body (#11) (hungarian.c, lines 28-66)
void init_potentials(int cost[N][N], int n, double dual_row[N], double dual_col[N]) {
init_potentials (#8)
init_potentials
init_potentials
Procedure Call Graph
main
hungarian_parallel
augmenting_path_search
init_potentials
optimality_check
Schedule Selection
Runtime Ratio
36.05 ms
Transformation Stack
Debug

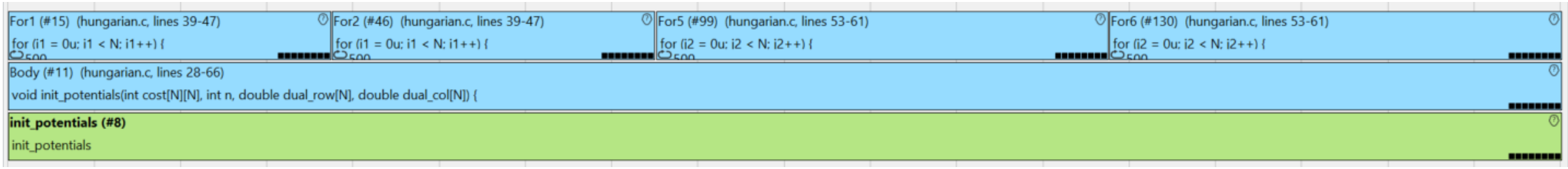

For1 (#15) (hungarian.c, lines 39-47)
for (i1 = 0u; i1 < N; i1++) {
1000
For3 (#57) (hungarian.c, lines 53-61)
for (i2 = 0u; i2 < N; i2++) {
1000
Body (#11) (hungarian.c, lines 28-66)
void init_potentials(int cost[N][N], int n, double dual_row[N], double dual_col[N]) {
init_potentials (#8)
init_potentials

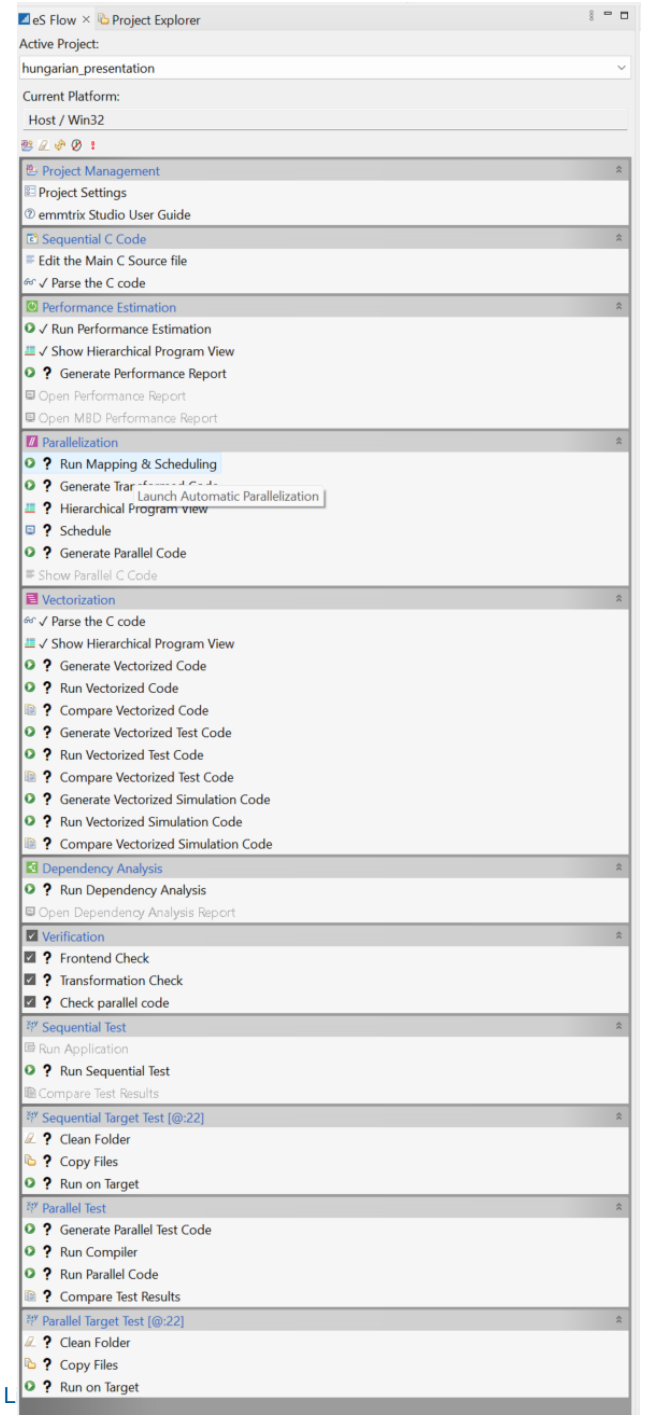

For1 (#15) (hungarian.c, lines 39-47)
for (i1 = 0u; i1 < N; i1++) {
Body (#11) (hungarian.c, lines 28-66)
void init_potentials(int cost[N][N], int n, double dual_row[N], double dual_col[N]) {
init_potentials (#8)
init_potentials
init_potentials (#8)
Assign core
Set core constraint
Set cluster constraint
Clear constraints
Clear constraints recursively
Create Transformation
Show subgraph
Show subgraph of parent
Show in Properties View
Advanced
Make main function...
BlockOutline...
Obfuscate...
ControlToGoto...
Control structure simplifier...
ConstPropagation...
Idioms recognition...
Loop Fusion...
Loop Unrolling...
VarSplit...
VarCoalescing...
Split array variables
PtrResolve...
InstructionSimplifier...
Extract MBD comments...
CodeSinking...
ExplicitCastInserter...
RemoveUnusedDecls...
DeadStmtRemoval...
Deroll...
MakeStatic...
Reorder...
RemoveGotos...
IfMerge...
BitBlast...
NormalizeInitializerLists...

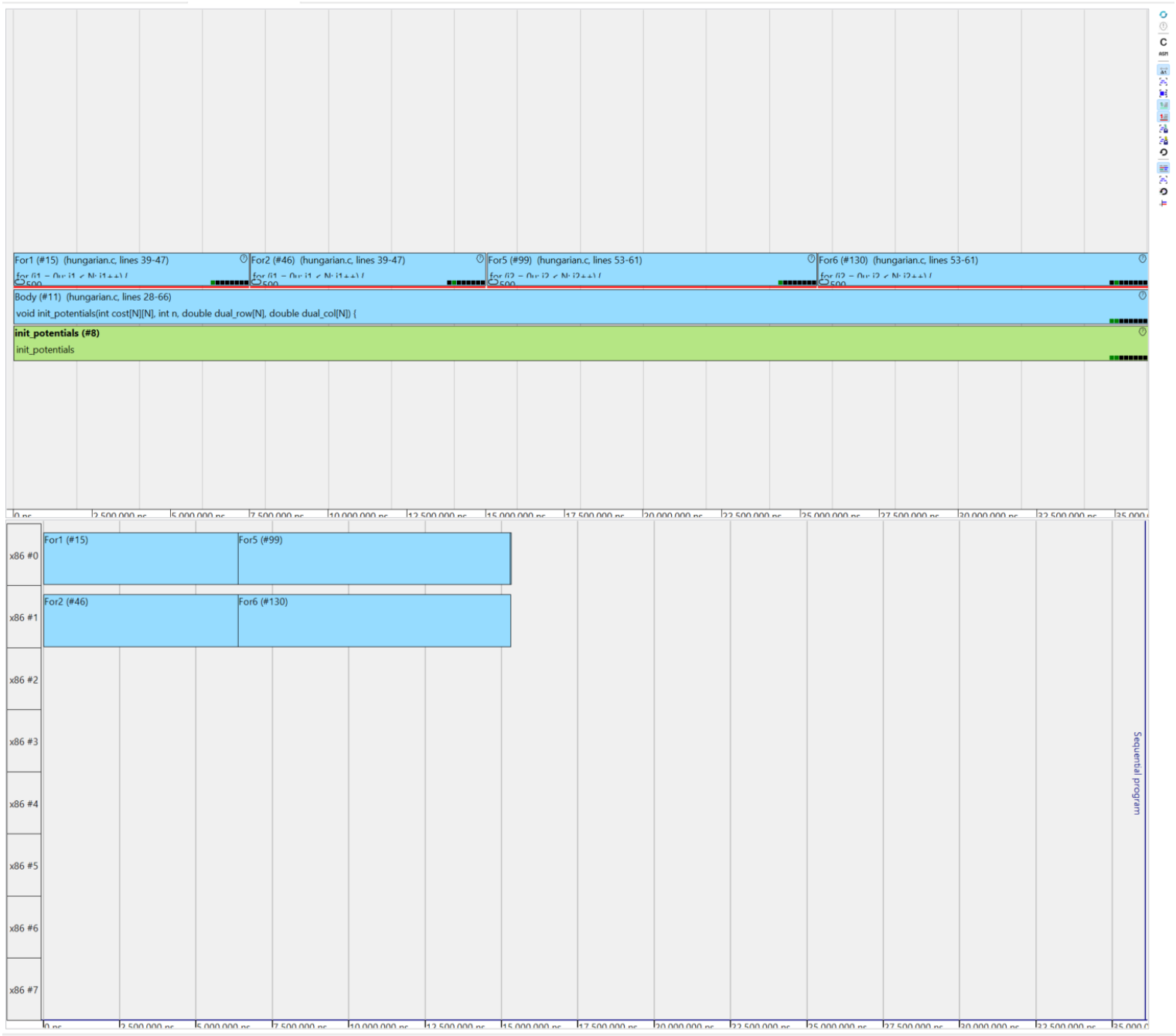

VarSplit: Split array variables
Available variables
double d_row[1000] - Dimension 0 - OK
double d_col[1000] - Dimension 0 - OK
int cost[1000][1000] - Dimension 0 - OK
int cost[1000][1000] - Dimension 1 - OK
double dual_row[1000] - Dimension 0 - OK
double dual_col[1000] - Dimension 0 - OK
Enable transformation
Variable name
d_row
Variable dimension
0
Split count
2
Perform loop splitting on loops containing the variable
set to Defau
set to Curre
Apply
OK
Cancel
For1 (#15) (hungarian.c, lines 39-47)
for (i1 = 0u; i1 < N; i1++) {
1000
Body (#11) (hungarian.c, lines 28-66)
void init_potentials(int cost[N][N], int n, double dual_row[N], double dual_col[N
init_potentials (#8)
init_potentials


For1 (#15) (hungarian.c, lines 39-47)
for (i1 = 0u; i1 < N; i1++) {
1000
For3 (#57) (hungarian.c, lines 53-61)
for (i2 = 0u; i2 < N; i2++) {
1000
Body (#11) (hungarian.c, lines 28-66)
void init_potentials(int cost[N][N], int n, double dual_row[N], double dual_col[N]) {
init_potentials (#8)
init_potentials
Body (#11)
Assign core
Set core constraint
Set cluster constraint
Clear constraints
Clear constraints recursively
Create Transformation
Show subgraph
Show subgraph of parent
Show in Properties View
Advanced
BlockOutline...
VarSplit...
Split array variables


VarSplit: Split array variables
Available variables
double d_row[1000] - Dimension 0 - OK
double d_col[1000] - Dimension 0 - OK
Enable transformation
Variable name
d_col
Variable dimension
0
Split count
2
Perform loop splitting on loops containing the variable
set to Defau
set to Curre
Apply
OK
Cancel
For1 (#15) (hungarian.c, lines 39-47)
for (i1 = 0u; i1 < N; i1++) {
1000
Body (#11) (hungarian.c, lines 28-66)
void init_potentials(int cost[N][N], int n, double dual_row[N], double dual_col[N])
init_potentials (#8)
init_potentials


For1 (#15) (hungarian.c, lines 39-47)
for (i1 = 0u; i1 < N; i1++) {
500
For2 (#46) (hungarian.c, lines 39-47)
for (i1 = 0u; i1 < N; i1++) {
500
For5 (#99) (hungarian.c, lines 53-61)
for (i2 = 0u; i2 < N; i2++) {
500
For6 (#130) (hungarian.c, lines 53-61)
for (i2 = 0u; i2 < N; i2++) {
500
Body (#11) (hungarian.c, lines 28-66)
void init_potentials(int cost[N][N], int n, double dual_row[N], double dual_col[N]) {
init_potentials (#8)
init_potentials


eS Flow
Project Explorer
Active Project:
hungarian_presentation
Current Platform:
Host / Win32
Project Management
Project Settings
emmtrix Studio User Guide
Sequential C Code
Edit the Main C Source file
✓ Parse the C code
Performance Estimation
✓ Run Performance Estimation
✓ Show Hierarchical Program View
Generate Performance Report
Open Performance Report
Open MBD Performance Report
Parallelization
Run Mapping & Scheduling
Launch Automatic Parallelization
Hierarchical Program View
Schedule
Generate Parallel Code
Show Parallel C Code
Vectorization
✓ Parse the C code
✓ Show Hierarchical Program View
Generate Vectorized Code
Run Vectorized Code
Compare Vectorized Code
Generate Vectorized Test Code
Run Vectorized Test Code
Compare Vectorized Test Code
Generate Vectorized Simulation Code
Run Vectorized Simulation Code
Compare Vectorized Simulation Code
Dependency Analysis
Run Dependency Analysis
Open Dependency Analysis Report
Verification
Frontend Check
Transformation Check
Check parallel code
Sequential Test
Run Application
Run Sequential Test
Compare Test Results
Sequential Target Test [@:22]
Clean Folder
Copy Files
Run on Target
Parallel Test
Generate Parallel Test Code
Run Compiler
Run Parallel Code
Compare Test Results
Parallel Target Test [@:22]
Clean Folder
Copy Files
Run on Target


For1 (#15) (hungarian.c, lines 39-47)
For2 (#46) (hungarian.c, lines 39-47)
For5 (#99) (hungarian.c, lines 53-61)
For6 (#130) (hungarian.c, lines 53-61)
Body (#11) (hungarian.c, lines 28-66)
void init_potentials(int cost[N][N], int n, double dual_row[N], double dual_col[N]) {
init_potentials (#8)
init_potentials
x86 #0
x86 #1
x86 #2
x86 #3
x86 #4
x86 #5
x86 #6
x86 #7
For1 (#15)
For5 (#99)
For2 (#46)
For6 (#130)
Sequential program


eS Flow
Project Explorer
Active Project:
hungarian_presentation
Current Platform:
Host / Win32
Project Management
Project Settings
emmtrix Studio User Guide
Sequential C Code
Edit the Main C Source file
✓ Parse the C code
Performance Estimation
✓ Run Performance Estimation
✓ Show Hierarchical Program View
Generate Performance Report
Open Performance Report
Open MBD Performance Report
Parallelization
✓ Run Mapping & Scheduling
Generate Transformed Code
✓ Hierarchical Program View
✓ Schedule
Generate Parallel Code
Show Parallel C Code
Vectorization
✓ Parse the C code
✓ Show Hierarchical Program View
Generate Vectorized Code
Run Vectorized Code
Compare Vectorized Code
Generate Vectorized Test Code
Run Vectorized Test Code
Compare Vectorized Test Code
Generate Vectorized Simulation Code
Run Vectorized Simulation Code
Compare Vectorized Simulation Code
Dependency Analysis
Run Dependency Analysis
Open Dependency Analysis Report
Verification
Frontend Check
Transformation Check
Check parallel code
Sequential Test
Run Application
Run Sequential Test
Compare Test Results
Sequential Target Test [@:22]
Clean Folder
Copy Files
Run on Target
Parallel Test
Generate Parallel Test Code
Run Compiler
Run Parallel Code
Compare Test Results
Parallel Target Test [@:22]
Clean Folder
Copy Files
Run on Target

ePS Implementation - Example

```
extern void init_potentials_p0(int cost_p0[100][100], int n_p0, double dual_row_p0[100], double dual_col_p0[100]) {
    size_t i1_p0;
    double d_row_split1_p0[50];
    size_t j1_p0;
    size_t i3_p0;
    double d_row_split2_p0[50];
    size_t j2_p0;
    double d_col_split1_p0[50];
    size_t i2_p0;
    size_t i4_p0;
    double d_col_split2_p0[50];

    for (i1_p0 = 0u; i1_p0 < 50; i1_p0 = i1_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
            if (temp_p0 > ((double)cost_p0[i1_p0][j1_p0])) {
                temp_p0 = (double)cost_p0[i1_p0][j1_p0];
            }
        }
        val2_p0 = temp_p0;
        d_row_split1_p0[i1_p0 - 0] = val2_p0;
    }

    for (i1_p0 = 50; i1_p0 < 100; i1_p0 = i1_p0 + 1) {
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
        }
    }

    for (i3_p0 = 0u; i3_p0 < 50; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split1_p0[i3_p0 - 0];
        dual_row_p0[i3_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 18, -1, d_row_split2_p0, 50 * sizeof(*d_row_split2_p0));

    for (i3_p0 = 50; i3_p0 < 100; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split2_p0[i3_p0 - 50];
        dual_row_p0[i3_p0] = val2_p0;
    }
```

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p0(int cost_p0[100][100], int n_p0, double dual_row_p0[100], double dual_col_p0[100]) {
    size_t i1_p0;
    double d_row_split1_p0[50];
    size_t j1_p0;
    size_t i3_p0;
    double d_row_split2_p0[50];
    size_t j2_p0;
    double d_col_split1_p0[50];
    size_t i2_p0;
    size_t i4_p0;
    double d_col_split2_p0[50];

    for (i1_p0 = 0u; i1_p0 < 50; i1_p0 = i1_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
            if (temp_p0 > ((double)cost_p0[i1_p0][j1_p0])) {
                temp_p0 = (double)cost_p0[i1_p0][j1_p0];
            }
        }
        val2_p0 = temp_p0;
        d_row_split1_p0[i1_p0 - 0] = val2_p0;
    }

    for (i1_p0 = 50; i1_p0 < 100; i1_p0 = i1_p0 + 1) {
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
        }
    }

    for (i3_p0 = 0u; i3_p0 < 50; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split1_p0[i3_p0 - 0];
        dual_row_p0[i3_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 18, -1, d_row_split2_p0, 50 * sizeof(*d_row_split2_p0));

    for (i3_p0 = 50; i3_p0 < 100; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split2_p0[i3_p0 - 50];
        dual_row_p0[i3_p0] = val2_p0;
    }
```

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p0(int cost_p0[100][100], int n_p0, double dual_row_p0[100], double dual_col_p0[100]) {
    size_t i1_p0;
    double d_row_split1_p0[50];
    size_t j1_p0;
    size_t i3_p0;
    double d_row_split2_p0[50];
    size_t j2_p0;
    double d_col_split1_p0[50];
    size_t i2_p0;
    size_t i4_p0;
    double d_col_split2_p0[50];

    for (i1_p0 = 0u; i1_p0 < 50; i1_p0 = i1_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
            if (temp_p0 > ((double)cost_p0[i1_p0][j1_p0])) {
                temp_p0 = (double)cost_p0[i1_p0][j1_p0];
            }
        }
        val2_p0 = temp_p0;
        d_row_split1_p0[i1_p0 - 0] = val2_p0;
    }

    for (i1_p0 = 50; i1_p0 < 100; i1_p0 = i1_p0 + 1) {
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
        }
    }

    for (i3_p0 = 0u; i3_p0 < 50; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split1_p0[i3_p0 - 0];
        dual_row_p0[i3_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 18, -1, d_row_split2_p0, 50 * sizeof(*d_row_split2_p0));

    for (i3_p0 = 50; i3_p0 < 100; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split2_p0[i3_p0 - 50];
        dual_row_p0[i3_p0] = val2_p0;
    }
```

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p0(int cost_p0[100][100], int n_p0, double dual_row_p0[100], double dual_col_p0[100]) {
    size_t i1_p0;
    double d_row_split1_p0[50];
    size_t j1_p0;
    size_t i3_p0;
    double d_row_split2_p0[50];
    size_t j2_p0;
    double d_col_split1_p0[50];
    size_t i2_p0;
    size_t i4_p0;
    double d_col_split2_p0[50];

    for (i1_p0 = 0u; i1_p0 < 50; i1_p0 = i1_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
            if (temp_p0 > ((double)cost_p0[i1_p0][j1_p0])) {
                temp_p0 = (double)cost_p0[i1_p0][j1_p0];
            }
        }
        val2_p0 = temp_p0;
        d_row_split1_p0[i1_p0 - 0] = val2_p0;
    }

    for (i1_p0 = 50; i1_p0 < 100; i1_p0 = i1_p0 + 1) {
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
        }
    }

    for (i3_p0 = 0u; i3_p0 < 50; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split1_p0[i3_p0 - 0];
        dual_row_p0[i3_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 18, -1, d_row_split2_p0, 50 * sizeof(*d_row_split2_p0));

    for (i3_p0 = 50; i3_p0 < 100; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split2_p0[i3_p0 - 50];
        dual_row_p0[i3_p0] = val2_p0;
    }
```

Init variables p0

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p0(int cost_p0[100][100], int n_p0, double dual_row_p0[100], double dual_col_p0[100]) {
    size_t i1_p0;
    double d_row_split1_p0[50];
    size_t j1_p0;
    size_t i3_p0;
    double d_row_split2_p0[50];
    size_t j2_p0;
    double d_col_split1_p0[50];
    size_t i2_p0;
    size_t i4_p0;
    double d_col_split2_p0[50];

    for (i1_p0 = 0u; i1_p0 < 50; i1_p0 = i1_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
            if (temp_p0 > ((double)cost_p0[i1_p0][j1_p0])) {
                temp_p0 = (double)cost_p0[i1_p0][j1_p0];
            }
        }
        val2_p0 = temp_p0;
        d_row_split1_p0[i1_p0 - 0] = val2_p0;
    }

    for (i1_p0 = 50; i1_p0 < 100; i1_p0 = i1_p0 + 1) {
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
        }
    }

    for (i3_p0 = 0u; i3_p0 < 50; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split1_p0[i3_p0 - 0];
        dual_row_p0[i3_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 18, -1, d_row_split2_p0, 50 * sizeof(*d_row_split2_p0));

    for (i3_p0 = 50; i3_p0 < 100; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split2_p0[i3_p0 - 50];
        dual_row_p0[i3_p0] = val2_p0;
    }
```

First phase p0

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p0(int cost_p0[100][100], int n_p0, double dual_row_p0[100], double dual_col_p0[100]) {
    size_t i1_p0;
    double d_row_split1_p0[50];
    size_t j1_p0;
    size_t i3_p0;
    double d_row_split2_p0[50];
    size_t j2_p0;
    double d_col_split1_p0[50];
    size_t i2_p0;
    size_t i4_p0;
    double d_col_split2_p0[50];

    for (i1_p0 = 0u; i1_p0 < 50; i1_p0 = i1_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
            if (temp_p0 > ((double)cost_p0[i1_p0][j1_p0])) {
                temp_p0 = (double)cost_p0[i1_p0][j1_p0];
            }
        }
        val2_p0 = temp_p0;
        d_row_split1_p0[i1_p0 - 0] = val2_p0;
    }

    for (i1_p0 = 50; i1_p0 < 100; i1_p0 = i1_p0 + 1) {
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
        }
    }

    for (i3_p0 = 0u; i3_p0 < 50; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split1_p0[i3_p0 - 0];
        dual_row_p0[i3_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 18, -1, d_row_split2_p0, 50 * sizeof(*d_row_split2_p0));

    for (i3_p0 = 50; i3_p0 < 100; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split2_p0[i3_p0 - 50];
        dual_row_p0[i3_p0] = val2_p0;
    }
```

???

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p0(int cost_p0[100][100], int n_p0, double dual_row_p0[100], double dual_col_p0[100]) {
    size_t i1_p0;
    double d_row_split1_p0[50];
    size_t j1_p0;
    size_t i3_p0;
    double d_row_split2_p0[50];
    size_t j2_p0;
    double d_col_split1_p0[50];
    size_t i2_p0;
    size_t i4_p0;
    double d_col_split2_p0[50];

    for (i1_p0 = 0u; i1_p0 < 50; i1_p0 = i1_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
            if (temp_p0 > ((double)cost_p0[i1_p0][j1_p0])) {
                temp_p0 = (double)cost_p0[i1_p0][j1_p0];
            }
        }
        val2_p0 = temp_p0;
        d_row_split1_p0[i1_p0 - 0] = val2_p0;
    }

    for (i1_p0 = 50; i1_p0 < 100; i1_p0 = i1_p0 + 1) {
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
        }
    }

    for (i3_p0 = 0u; i3_p0 < 50; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split1_p0[i3_p0 - 0];
        dual_row_p0[i3_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 18, -1, d_row_split2_p0, 50 * sizeof(*d_row_split2_p0));

    for (i3_p0 = 50; i3_p0 < 100; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split2_p0[i3_p0 - 50];
        dual_row_p0[i3_p0] = val2_p0;
    }
```

Merge partial results of p0

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p0(int cost_p0[100][100], int n_p0, double dual_row_p0[100], double dual_col_p0[100]) {
    size_t i1_p0;
    double d_row_split1_p0[50];
    size_t j1_p0;
    size_t i3_p0;
    double d_row_split2_p0[50];
    size_t j2_p0;
    double d_col_split1_p0[50];
    size_t i2_p0;
    size_t i4_p0;
    double d_col_split2_p0[50];

    for (i1_p0 = 0u; i1_p0 < 50; i1_p0 = i1_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
            if (temp_p0 > ((double)cost_p0[i1_p0][j1_p0])) {
                temp_p0 = (double)cost_p0[i1_p0][j1_p0];
            }
        }
        val2_p0 = temp_p0;
        d_row_split1_p0[i1_p0 - 0] = val2_p0;
    }

    for (i1_p0 = 50; i1_p0 < 100; i1_p0 = i1_p0 + 1) {
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
        }
    }

    for (i3_p0 = 0u; i3_p0 < 50; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split1_p0[i3_p0 - 0];
        dual_row_p0[i3_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 18, -1, d_row_split2_p0, 50 * sizeof(*d_row_split2_p0));

    for (i3_p0 = 50; i3_p0 < 100; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split2_p0[i3_p0 - 50];
        dual_row_p0[i3_p0] = val2_p0;
    }
```

Receive partial results from p1

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p0(int cost_p0[100][100], int n_p0, double dual_row_p0[100], double dual_col_p0[100]) {
    size_t i1_p0;
    double d_row_split1_p0[50];
    size_t j1_p0;
    size_t i3_p0;
    double d_row_split2_p0[50];
    size_t j2_p0;
    double d_col_split1_p0[50];
    size_t i2_p0;
    size_t i4_p0;
    double d_col_split2_p0[50];

    for (i1_p0 = 0u; i1_p0 < 50; i1_p0 = i1_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
            if (temp_p0 > ((double)cost_p0[i1_p0][j1_p0])) {
                temp_p0 = (double)cost_p0[i1_p0][j1_p0];
            }
        }
        val2_p0 = temp_p0;
        d_row_split1_p0[i1_p0 - 0] = val2_p0;
    }

    for (i1_p0 = 50; i1_p0 < 100; i1_p0 = i1_p0 + 1) {
        for (j1_p0 = 0u; j1_p0 < 100; j1_p0 = j1_p0 + 1) {
        }
    }

    for (i3_p0 = 0u; i3_p0 < 50; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split1_p0[i3_p0 - 0];
        dual_row_p0[i3_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 18, -1, d_row_split2_p0, 50 * sizeof(*d_row_split2_p0));

    for (i3_p0 = 50; i3_p0 < 100; i3_p0 = i3_p0 + 1) {
        double val2_p0;
        val2_p0 = d_row_split2_p0[i3_p0 - 50];
        dual_row_p0[i3_p0] = val2_p0;
    }
```

Merge partial results of p1

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
    for (j2_p0 = 0u; j2_p0 < 50; j2_p0 = j2_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (i2_p0 = 0u; i2_p0 < 100; i2_p0 = i2_p0 + 1) {
            if (temp_p0 > (cost_p0[i2_p0][j2_p0] - dual_row_p0[i2_p0])) {
                temp_p0 = cost_p0[i2_p0][j2_p0] - dual_row_p0[i2_p0];
            }
        }

        val2_p0 = temp_p0;
        d_col_split1_p0[j2_p0 - 0] = val2_p0;
    }

    EMX_Send(0, 1, 17, -1, dual_row_p0, 100 * sizeof(*dual_row_p0));

    for (j2_p0 = 50; j2_p0 < 100; j2_p0 = j2_p0 + 1) {
        for (i2_p0 = 0u; i2_p0 < 100; i2_p0 = i2_p0 + 1) {
        }
    }

    for (i4_p0 = 0u; i4_p0 < 50; i4_p0 = i4_p0 + 1) {
        double val2_p0;

        val2_p0 = d_col_split1_p0[i4_p0 - 0];
        dual_col_p0[i4_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 19, -1, d_col_split2_p0, 50 * sizeof(*d_col_split2_p0));

    for (i4_p0 = 50; i4_p0 < 100; i4_p0 = i4_p0 + 1) {
        double val2_p0;
        val2_p0 = d_col_split2_p0[i4_p0 - 50];
        dual_col_p0[i4_p0] = val2_p0;
    }
}
```

Second phase p0

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
    for (j2_p0 = 0u; j2_p0 < 50; j2_p0 = j2_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (i2_p0 = 0u; i2_p0 < 100; i2_p0 = i2_p0 + 1) {
            if (temp_p0 > (cost_p0[i2_p0][j2_p0] - dual_row_p0[i2_p0])) {
                temp_p0 = cost_p0[i2_p0][j2_p0] - dual_row_p0[i2_p0];
            }
        }

        val2_p0 = temp_p0;
        d_col_split1_p0[j2_p0 - 0] = val2_p0;
    }

    EMX_Send(0, 1, 17, -1, dual_row_p0, 100 * sizeof(*dual_row_p0));

    for (j2_p0 = 50; j2_p0 < 100; j2_p0 = j2_p0 + 1) {
        for (i2_p0 = 0u; i2_p0 < 100; i2_p0 = i2_p0 + 1) {
        }
    }

    for (i4_p0 = 0u; i4_p0 < 50; i4_p0 = i4_p0 + 1) {
        double val2_p0;

        val2_p0 = d_col_split1_p0[i4_p0 - 0];
        dual_col_p0[i4_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 19, -1, d_col_split2_p0, 50 * sizeof(*d_col_split2_p0));

    for (i4_p0 = 50; i4_p0 < 100; i4_p0 = i4_p0 + 1) {
        double val2_p0;
        val2_p0 = d_col_split2_p0[i4_p0 - 50];
        dual_col_p0[i4_p0] = val2_p0;
    }
}
```

Send dual_row (dependency in second phase) to p1

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
    for (j2_p0 = 0u; j2_p0 < 50; j2_p0 = j2_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (i2_p0 = 0u; i2_p0 < 100; i2_p0 = i2_p0 + 1) {
            if (temp_p0 > (cost_p0[i2_p0][j2_p0] - dual_row_p0[i2_p0])) {
                temp_p0 = cost_p0[i2_p0][j2_p0] - dual_row_p0[i2_p0];
            }
        }

        val2_p0 = temp_p0;
        d_col_split1_p0[j2_p0 - 0] = val2_p0;
    }

    EMX_Send(0, 1, 17, -1, dual_row_p0, 100 * sizeof(*dual_row_p0));

    for (j2_p0 = 50; j2_p0 < 100; j2_p0 = j2_p0 + 1) {
        for (i2_p0 = 0u; i2_p0 < 100; i2_p0 = i2_p0 + 1) {
        }
    }

    for (i4_p0 = 0u; i4_p0 < 50; i4_p0 = i4_p0 + 1) {
        double val2_p0;

        val2_p0 = d_col_split1_p0[i4_p0 - 0];
        dual_col_p0[i4_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 19, -1, d_col_split2_p0, 50 * sizeof(*d_col_split2_p0));

    for (i4_p0 = 50; i4_p0 < 100; i4_p0 = i4_p0 + 1) {
        double val2_p0;
        val2_p0 = d_col_split2_p0[i4_p0 - 50];
        dual_col_p0[i4_p0] = val2_p0;
    }
}
```

???

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
    for (j2_p0 = 0u; j2_p0 < 50; j2_p0 = j2_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (i2_p0 = 0u; i2_p0 < 100; i2_p0 = i2_p0 + 1) {
            if (temp_p0 > (cost_p0[i2_p0][j2_p0] - dual_row_p0[i2_p0])) {
                temp_p0 = cost_p0[i2_p0][j2_p0] - dual_row_p0[i2_p0];
            }
        }

        val2_p0 = temp_p0;
        d_col_split1_p0[j2_p0 - 0] = val2_p0;
    }

    EMX_Send(0, 1, 17, -1, dual_row_p0, 100 * sizeof(*dual_row_p0));

    for (j2_p0 = 50; j2_p0 < 100; j2_p0 = j2_p0 + 1) {
        for (i2_p0 = 0u; i2_p0 < 100; i2_p0 = i2_p0 + 1) {
        }
    }

    for (i4_p0 = 0u; i4_p0 < 50; i4_p0 = i4_p0 + 1) {
        double val2_p0;

        val2_p0 = d_col_split1_p0[i4_p0 - 0];
        dual_col_p0[i4_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 19, -1, d_col_split2_p0, 50 * sizeof(*d_col_split2_p0));

    for (i4_p0 = 50; i4_p0 < 100; i4_p0 = i4_p0 + 1) {
        double val2_p0;
        val2_p0 = d_col_split2_p0[i4_p0 - 50];
        dual_col_p0[i4_p0] = val2_p0;
    }
}
```

Merge partial results of p0

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
    for (j2_p0 = 0u; j2_p0 < 50; j2_p0 = j2_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (i2_p0 = 0u; i2_p0 < 100; i2_p0 = i2_p0 + 1) {
            if (temp_p0 > (cost_p0[i2_p0][j2_p0] - dual_row_p0[i2_p0])) {
                temp_p0 = cost_p0[i2_p0][j2_p0] - dual_row_p0[i2_p0];
            }
        }

        val2_p0 = temp_p0;
        d_col_split1_p0[j2_p0 - 0] = val2_p0;
    }

    EMX_Send(0, 1, 17, -1, dual_row_p0, 100 * sizeof(*dual_row_p0));

    for (j2_p0 = 50; j2_p0 < 100; j2_p0 = j2_p0 + 1) {
        for (i2_p0 = 0u; i2_p0 < 100; i2_p0 = i2_p0 + 1) {
        }
    }

    for (i4_p0 = 0u; i4_p0 < 50; i4_p0 = i4_p0 + 1) {
        double val2_p0;

        val2_p0 = d_col_split1_p0[i4_p0 - 0];
        dual_col_p0[i4_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 19, -1, d_col_split2_p0, 50 * sizeof(*d_col_split2_p0));

    for (i4_p0 = 50; i4_p0 < 100; i4_p0 = i4_p0 + 1) {
        double val2_p0;
        val2_p0 = d_col_split2_p0[i4_p0 - 50];
        dual_col_p0[i4_p0] = val2_p0;
    }
}
```

Receive partial results from p1

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
    for (j2_p0 = 0u; j2_p0 < 50; j2_p0 = j2_p0 + 1) {
        double temp_p0 = 0x7fffffff;
        double val2_p0;
        for (i2_p0 = 0u; i2_p0 < 100; i2_p0 = i2_p0 + 1) {
            if (temp_p0 > (cost_p0[i2_p0][j2_p0] - dual_row_p0[i2_p0])) {
                temp_p0 = cost_p0[i2_p0][j2_p0] - dual_row_p0[i2_p0];
            }
        }

        val2_p0 = temp_p0;
        d_col_split1_p0[j2_p0 - 0] = val2_p0;
    }

    EMX_Send(0, 1, 17, -1, dual_row_p0, 100 * sizeof(*dual_row_p0));

    for (j2_p0 = 50; j2_p0 < 100; j2_p0 = j2_p0 + 1) {
        for (i2_p0 = 0u; i2_p0 < 100; i2_p0 = i2_p0 + 1) {
        }
    }

    for (i4_p0 = 0u; i4_p0 < 50; i4_p0 = i4_p0 + 1) {
        double val2_p0;

        val2_p0 = d_col_split1_p0[i4_p0 - 0];
        dual_col_p0[i4_p0] = val2_p0;
    }

    EMX_Recv(1, 0, 19, -1, d_col_split2_p0, 50 * sizeof(*d_col_split2_p0));

    for (i4_p0 = 50; i4_p0 < 100; i4_p0 = i4_p0 + 1) {
        double val2_p0;
        val2_p0 = d_col_split2_p0[i4_p0 - 50];
        dual_col_p0[i4_p0] = val2_p0;
    }
}
```

Merge partial results of p1

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p1(int cost_p1[100][100]) {
    double dual_row_p1[100];
    size_t i1_p1;
    double d_row_split2_p1[50];
    size_t j1_p1;
    size_t j2_p1;
    double d_col_split2_p1[50];
    size_t i2_p1;

    for (i1_p1 = 50; i1_p1 < 100; i1_p1 = i1_p1 + 1) {
        double temp_p1 = 0x7fffffff;
        double val2_p1;

        for (j1_p1 = 0u; j1_p1 < 100; j1_p1 = j1_p1 + 1) {
            if (temp_p1 > ((double)cost_p1[i1_p1][j1_p1])) {
                temp_p1 = (double)cost_p1[i1_p1][j1_p1];
            }
        }

        val2_p1 = temp_p1;
        d_row_split2_p1[i1_p1 - 50] = val2_p1;

    }

    EMX_Send(1, 0, 18, -1, d_row_split2_p1, 50 * sizeof(*d_row_split2_p1));

    EMX_Recv(0, 1, 17, -1, dual_row_p1, 100 * sizeof(*dual_row_p1));

    for (j2_p1 = 50; j2_p1 < 100; j2_p1 = j2_p1 + 1) {
        double temp_p1 = 0x7fffffff;
        double val2_p1;
        for (i2_p1 = 0u; i2_p1 < 100; i2_p1 = i2_p1 + 1) {
            if (temp_p1 > (cost_p1[i2_p1][j2_p1] - dual_row_p1[i2_p1])) {
                temp_p1 = cost_p1[i2_p1][j2_p1] - dual_row_p1[i2_p1];
            }
        }
        val2_p1 = temp_p1;
        d_col_split2_p1[j2_p1 - 50] = val2_p1;
    }
    EMX_Send(1, 0, 19, -1, d_col_split2_p1, 50 * sizeof(*d_col_split2_p1));
}
```

Init variables p1

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p1(int cost_p1[100][100]) {
    double dual_row_p1[100];
    size_t i1_p1;
    double d_row_split2_p1[50];
    size_t j1_p1;
    size_t j2_p1;
    double d_col_split2_p1[50];
    size_t i2_p1;

    for (i1_p1 = 50; i1_p1 < 100; i1_p1 = i1_p1 + 1) {
        double temp_p1 = 0x7fffffff;
        double val2_p1;

        for (j1_p1 = 0u; j1_p1 < 100; j1_p1 = j1_p1 + 1) {
            if (temp_p1 > ((double)cost_p1[i1_p1][j1_p1])) {
                temp_p1 = (double)cost_p1[i1_p1][j1_p1];
            }
        }

        val2_p1 = temp_p1;
        d_row_split2_p1[i1_p1 - 50] = val2_p1;

    }

    EMX_Send(1, 0, 18, -1, d_row_split2_p1, 50 * sizeof(*d_row_split2_p1));

    EMX_Recv(0, 1, 17, -1, dual_row_p1, 100 * sizeof(*dual_row_p1));

    for (j2_p1 = 50; j2_p1 < 100; j2_p1 = j2_p1 + 1) {
        double temp_p1 = 0x7fffffff;
        double val2_p1;
        for (i2_p1 = 0u; i2_p1 < 100; i2_p1 = i2_p1 + 1) {
            if (temp_p1 > (cost_p1[i2_p1][j2_p1] - dual_row_p1[i2_p1])) {
                temp_p1 = cost_p1[i2_p1][j2_p1] - dual_row_p1[i2_p1];
            }
        }
        val2_p1 = temp_p1;
        d_col_split2_p1[j2_p1 - 50] = val2_p1;
    }
    EMX_Send(1, 0, 19, -1, d_col_split2_p1, 50 * sizeof(*d_col_split2_p1));
}
```

First phase p1

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p1(int cost_p1[100][100]) {
    double dual_row_p1[100];
    size_t i1_p1;
    double d_row_split2_p1[50];
    size_t j1_p1;
    size_t j2_p1;
    double d_col_split2_p1[50];
    size_t i2_p1;

    for (i1_p1 = 50; i1_p1 < 100; i1_p1 = i1_p1 + 1) {
        double temp_p1 = 0x7fffffff;
        double val2_p1;

        for (j1_p1 = 0u; j1_p1 < 100; j1_p1 = j1_p1 + 1) {
            if (temp_p1 > ((double)cost_p1[i1_p1][j1_p1])) {
                temp_p1 = (double)cost_p1[i1_p1][j1_p1];
            }
        }

        val2_p1 = temp_p1;
        d_row_split2_p1[i1_p1 - 50] = val2_p1;

    }
    EMX_Send(1, 0, 18, -1, d_row_split2_p1, 50 * sizeof(*d_row_split2_p1));

    EMX_Recv(0, 1, 17, -1, dual_row_p1, 100 * sizeof(*dual_row_p1));

    for (j2_p1 = 50; j2_p1 < 100; j2_p1 = j2_p1 + 1) {
        double temp_p1 = 0x7fffffff;
        double val2_p1;
        for (i2_p1 = 0u; i2_p1 < 100; i2_p1 = i2_p1 + 1) {
            if (temp_p1 > (cost_p1[i2_p1][j2_p1] - dual_row_p1[i2_p1])) {
                temp_p1 = cost_p1[i2_p1][j2_p1] - dual_row_p1[i2_p1];
            }
        }
        val2_p1 = temp_p1;
        d_col_split2_p1[j2_p1 - 50] = val2_p1;
    }
    EMX_Send(1, 0, 19, -1, d_col_split2_p1, 50 * sizeof(*d_col_split2_p1));
}
```

Send partial results to p0

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p1(int cost_p1[100][100]) {
    double dual_row_p1[100];
    size_t i1_p1;
    double d_row_split2_p1[50];
    size_t j1_p1;
    size_t j2_p1;
    double d_col_split2_p1[50];
    size_t i2_p1;

    for (i1_p1 = 50; i1_p1 < 100; i1_p1 = i1_p1 + 1) {
        double temp_p1 = 0x7fffffff;
        double val2_p1;

        for (j1_p1 = 0u; j1_p1 < 100; j1_p1 = j1_p1 + 1) {
            if (temp_p1 > ((double)cost_p1[i1_p1][j1_p1])) {
                temp_p1 = (double)cost_p1[i1_p1][j1_p1];
            }
        }

        val2_p1 = temp_p1;
        d_row_split2_p1[i1_p1 - 50] = val2_p1;

    }

    EMX_Send(1, 0, 18, -1, d_row_split2_p1, 50 * sizeof(*d_row_split2_p1));

    EMX_Recv(0, 1, 17, -1, dual_row_p1, 100 * sizeof(*dual_row_p1));

    for (j2_p1 = 50; j2_p1 < 100; j2_p1 = j2_p1 + 1) {
        double temp_p1 = 0x7fffffff;
        double val2_p1;
        for (i2_p1 = 0u; i2_p1 < 100; i2_p1 = i2_p1 + 1) {
            if (temp_p1 > (cost_p1[i2_p1][j2_p1] - dual_row_p1[i2_p1])) {
                temp_p1 = cost_p1[i2_p1][j2_p1] - dual_row_p1[i2_p1];
            }
        }
        val2_p1 = temp_p1;
        d_col_split2_p1[j2_p1 - 50] = val2_p1;
    }
    EMX_Send(1, 0, 19, -1, d_col_split2_p1, 50 * sizeof(*d_col_split2_p1));
}
```

Receive dual_row (dependency for second phase) from p0

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p1(int cost_p1[100][100]) {
    double dual_row_p1[100];
    size_t i1_p1;
    double d_row_split2_p1[50];
    size_t j1_p1;
    size_t j2_p1;
    double d_col_split2_p1[50];
    size_t i2_p1;

    for (i1_p1 = 50; i1_p1 < 100; i1_p1 = i1_p1 + 1) {
        double temp_p1 = 0x7fffffff;
        double val2_p1;

        for (j1_p1 = 0u; j1_p1 < 100; j1_p1 = j1_p1 + 1) {
            if (temp_p1 > ((double)cost_p1[i1_p1][j1_p1])) {
                temp_p1 = (double)cost_p1[i1_p1][j1_p1];
            }
        }

        val2_p1 = temp_p1;
        d_row_split2_p1[i1_p1 - 50] = val2_p1;

    }

    EMX_Send(1, 0, 18, -1, d_row_split2_p1, 50 * sizeof(*d_row_split2_p1));

    EMX_Recv(0, 1, 17, -1, dual_row_p1, 100 * sizeof(*dual_row_p1));

    for (j2_p1 = 50; j2_p1 < 100; j2_p1 = j2_p1 + 1) {
        double temp_p1 = 0x7fffffff;
        double val2_p1;
        for (i2_p1 = 0u; i2_p1 < 100; i2_p1 = i2_p1 + 1) {
            if (temp_p1 > (cost_p1[i2_p1][j2_p1] - dual_row_p1[i2_p1])) {
                temp_p1 = cost_p1[i2_p1][j2_p1] - dual_row_p1[i2_p1];
            }
        }
        val2_p1 = temp_p1;
        d_col_split2_p1[j2_p1 - 50] = val2_p1;
    }
    EMX_Send(1, 0, 19, -1, d_col_split2_p1, 50 * sizeof(*d_col_split2_p1));
}
```

Second phase p1

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

ePS Implementation - Example

```
extern void init_potentials_p1(int cost_p1[100][100]) {
    double dual_row_p1[100];
    size_t i1_p1;
    double d_row_split2_p1[50];
    size_t j1_p1;
    size_t j2_p1;
    double d_col_split2_p1[50];
    size_t i2_p1;

    for (i1_p1 = 50; i1_p1 < 100; i1_p1 = i1_p1 + 1) {
        double temp_p1 = 0x7fffffff;
        double val2_p1;

        for (j1_p1 = 0u; j1_p1 < 100; j1_p1 = j1_p1 + 1) {
            if (temp_p1 > ((double)cost_p1[i1_p1][j1_p1])) {
                temp_p1 = (double)cost_p1[i1_p1][j1_p1];
            }
        }

        val2_p1 = temp_p1;
        d_row_split2_p1[i1_p1 - 50] = val2_p1;

    }

    EMX_Send(1, 0, 18, -1, d_row_split2_p1, 50 * sizeof(*d_row_split2_p1));

    EMX_Recv(0, 1, 17, -1, dual_row_p1, 100 * sizeof(*dual_row_p1));

    for (j2_p1 = 50; j2_p1 < 100; j2_p1 = j2_p1 + 1) {
        double temp_p1 = 0x7fffffff;
        double val2_p1;
        for (i2_p1 = 0u; i2_p1 < 100; i2_p1 = i2_p1 + 1) {
            if (temp_p1 > (cost_p1[i2_p1][j2_p1] - dual_row_p1[i2_p1])) {
                temp_p1 = cost_p1[i2_p1][j2_p1] - dual_row_p1[i2_p1];
            }
        }
        val2_p1 = temp_p1;
        d_col_split2_p1[j2_p1 - 50] = val2_p1;
    }
    EMX_Send(1, 0, 19, -1, d_col_split2_p1, 50 * sizeof(*d_col_split2_p1));
}
```

Send partial results to p0

Parallel code characteristics:

- Each function and each variable are duplicated for each processor
- Fixed to exactly one problem size
- Fixed to exactly one processer count
- Added synchronization and communication

Test Environment

AMD Ryzen 9 7940HS

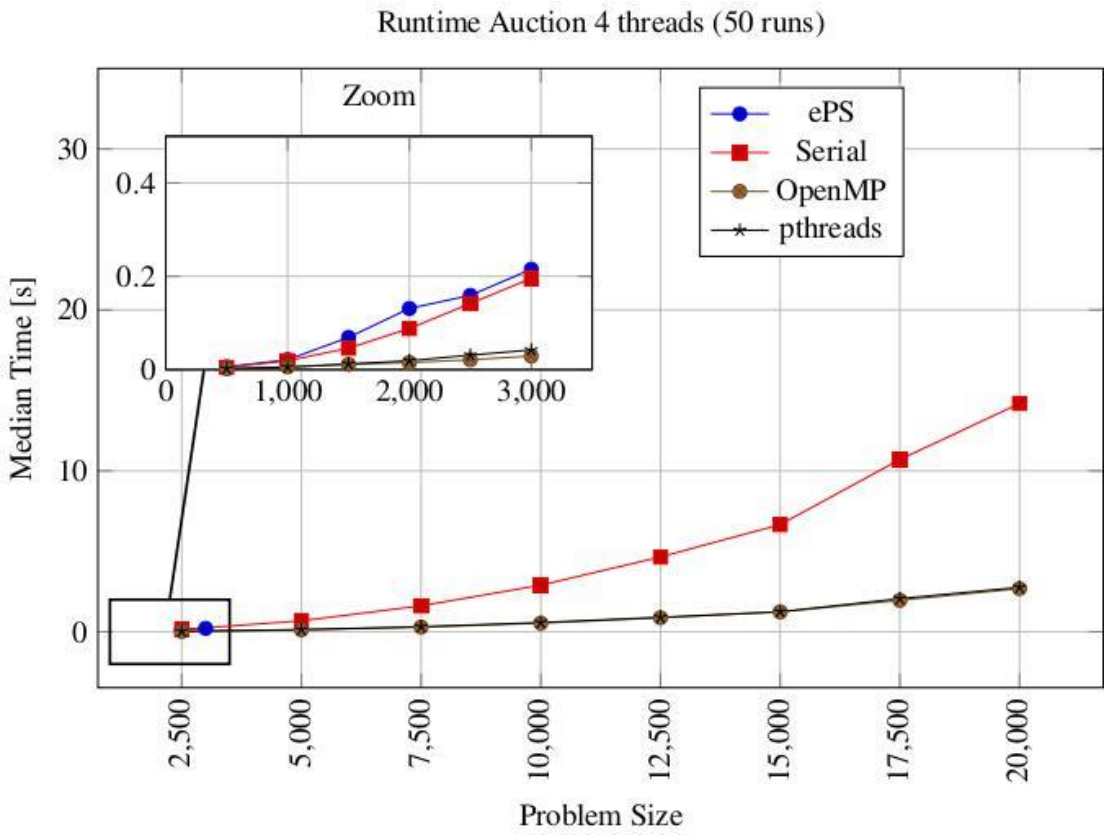
- 8 physical cores
- 16 logical cores

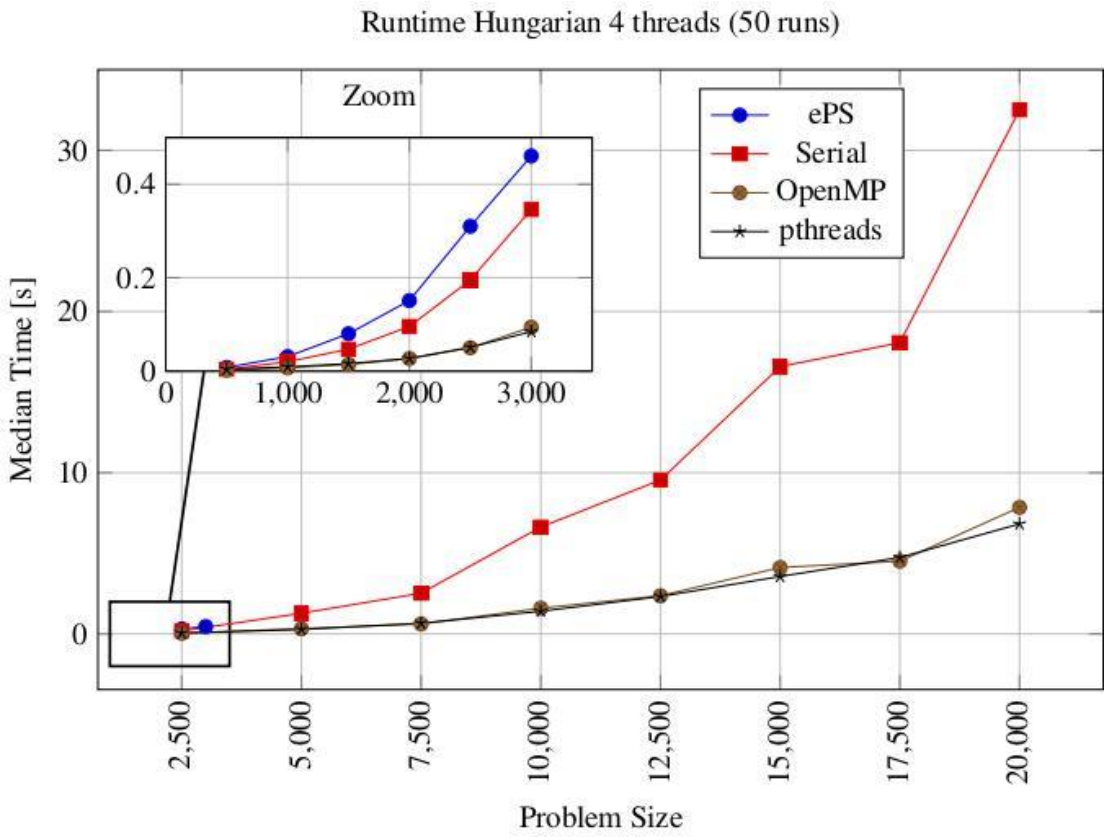
Ubuntu 24.04.2 LTS

Compiler

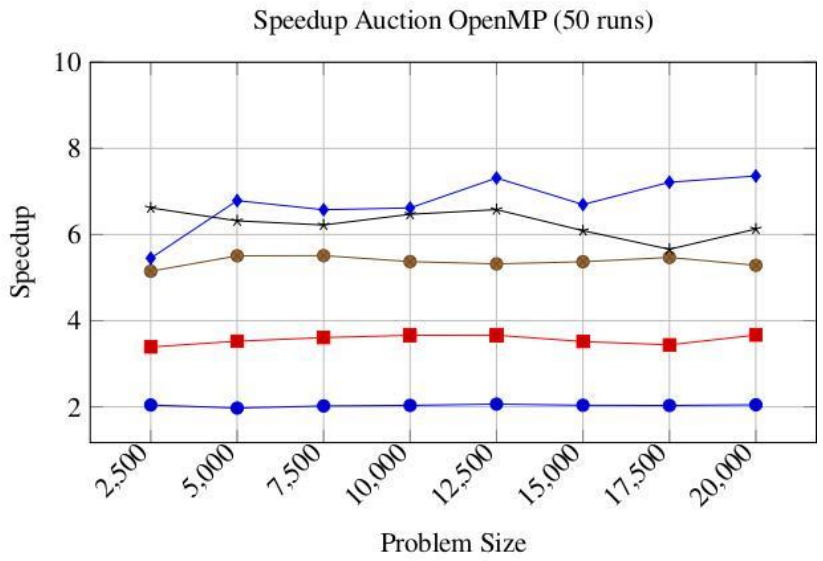
- C++ with g++, ePS-generated C code with gcc
- libgomp and glibc runtime libraries
- -O3 optimization flag

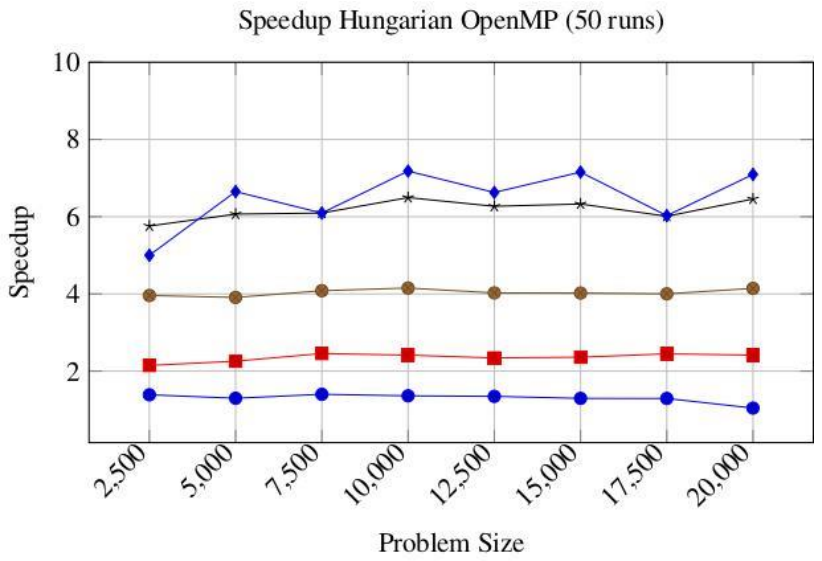
Performance

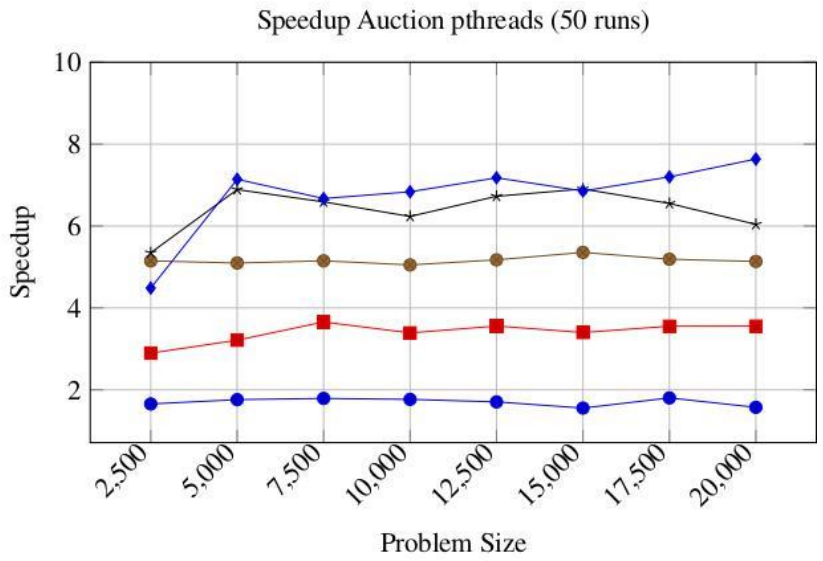


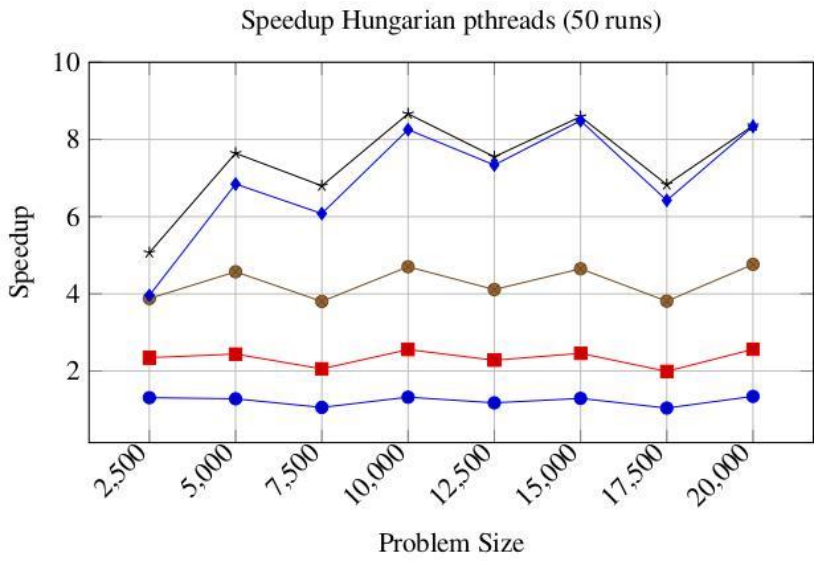


Scalability

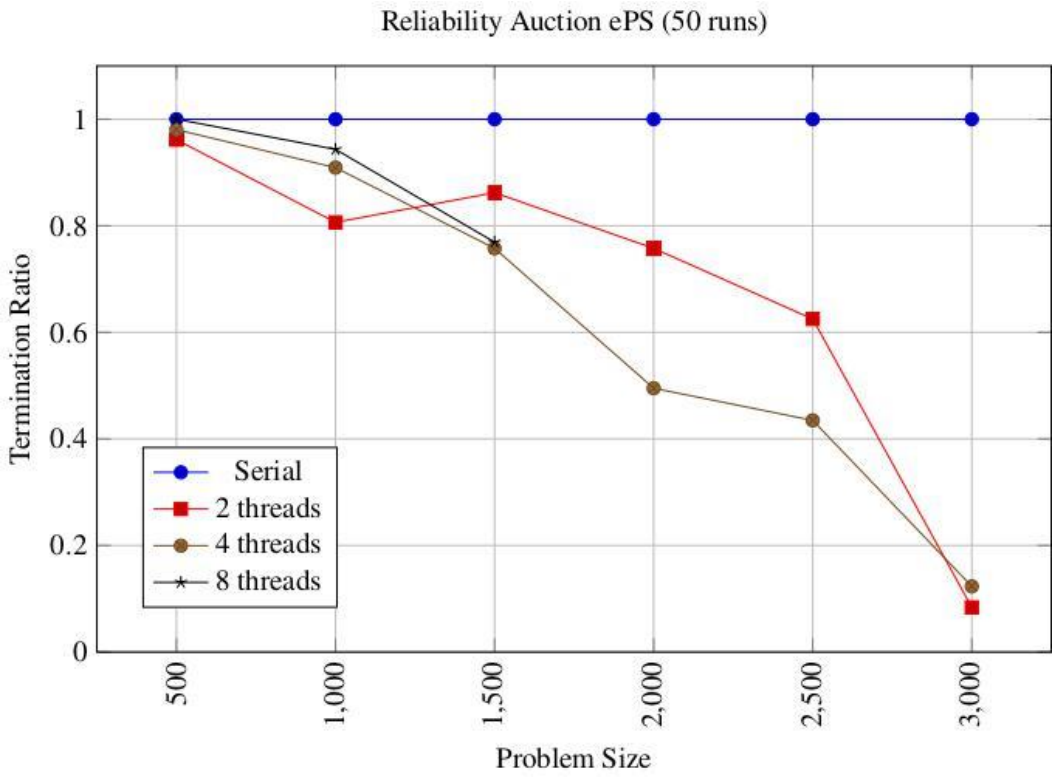


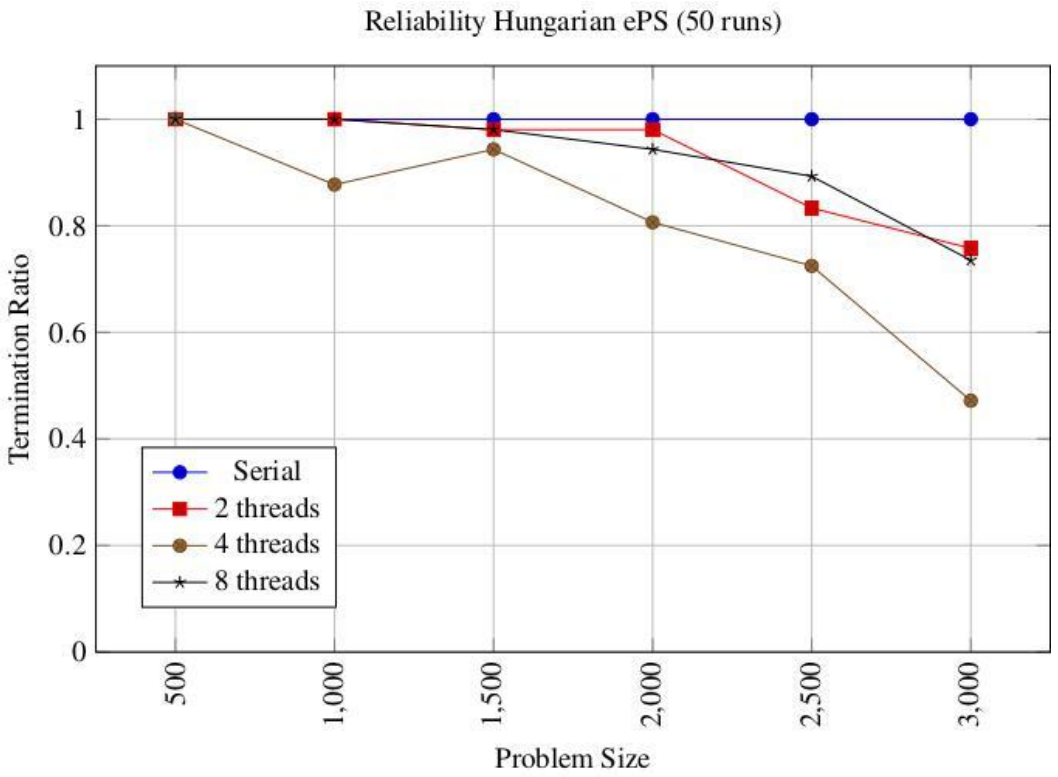






Reliability - ePS





Development Cost

	pthreads	OpenMP API	ePS
Development effort			
Expertise needed			
Refactoring			
Maintenance			
Net cost			

Conclusion

01

OpenMP and pthreads are mature, reliable solutions

02

OpenMP: excels in usability and maintainability

03

pthreads: provides fine-grained control and equivalent performance to OpenMP

04

ePS (emmtrixPS): underperformed across most evaluation criteria — making it unsuitable in its present form

Recommendation

OpenMP for most projects prioritizing development speed, maintainability, and predictable scaling.

Pthreads when you need low-level control and your team can manage the extra complexity.

Don’t adopt ePS for production parallelization until improvements in reliability, performance, and usability are made.

Thank you for your attention.

Questions?

Sources

[1] emmtrix Technologies GmbH: Automation Tools for Better Code

[2] Dimitri P. Bertsekas and David A. Castañon: Parallel Synchronous and Asynchronous Implementations of the Auction Algorithm

[3] Date, Ketan and Nagi, Rakesh: GPU-Accelerated Hungarian Algorithms for the Linear Assignment Problem