A Parallel Biobjective Shortest Path Algorithm

F. Antonio Medrano and Richard L. Church

Project 301CR, GeoTrans Report 2014-12-01
December 2014

Photos courtesy of DOE/NREL

University of California, Santa Barbara
Department of Geography
1832 Ellison Hall
Santa Barbara, CA 93106
medrano@geog.ucsb.edu
church@geog.ucsb.edu

A Parallel Biobjective Shortest Path Algorithm

F. Antonio Medrano
Richard L. Church

Corridor Location Project
GeoTrans Laboratory
Department of Geography
University of California, Santa Barbara
December 31, 2014

This report has been developed as a part of the corridor location research project at the University of California, Santa Barbara. The goal of this project is to take a fresh look at the process of corridor location, and develop a set of algorithms that compute path alternatives using a foundation of solid geographical theory in order to offer designers better tools for developing quality alternatives that consider the entire spectrum of viable solutions. And just as importantly, as data sets become increasingly massive and present challenging computational elements, it is important that algorithms be efficient and able to take advantage of parallel computing resources. Please cite this report as: Medrano, FA, and RL Church (2014) "A Parallel Biobjective Shortest Path Algorithm" (Report #12-14-01), GeoTrans Laboratory, UCSB, Santa Barbara CA.

I. Introduction

Exponential growth in the capabilities of computerized data collection and analysis over the past few decades has resulted in the availability of massive data sets and networks for modeling and simulation. Traditional problems of public systems development such as corridor location for new transmission lines, pipelines, roadways and railways have always been considered a wicked optimization problem (Liebman 1976), and are now even more complicated given higher resolutions of satellite imagery for generating finer grained terrain network models. New frontiers in the analysis of large network data sets include the study of relationships between social media users (1.23 billion active Facebook users as of January 2014), and grouping by attributes within large online data repositories (Flickr contained over 8 billion photographs as of March 2013, a large portion of which are geotagged). These, and countless other recent data sources have served as the impetus for new terminology such as *big data* for working with data sets far too large to be processed by traditional database management tools, and the field of *analytics* for discovering meaningful results from these overwhelmingly large data sets.

As data sets increase in size, the computation required to do meaningful analysis on the data also increases. Moore's Law (Moore 1965) states that the number of transistors capable of being placed in an integrated circuit, and thus the computational power of a CPU, doubles every two years. This rule has held true since its inception in 1965 and until the early 2000's was mostly realized through faster processor clock speeds. In 2004, thermal limitations prevented any further increase in processor clock speeds, creating a paradigm shift from faster clocks to multiple processor cores per CPU. Legacy programming code though cannot take advantage of multiple cores, and requires extensive rewrites to a parallel language in order to use the full capabilities of modern computers. This is not a simple task, as parallel computing introduces problems such as race conditions and deadlocks, which can result in non-deterministic behavior, infinite loops, or runtime failures. Proper implementation of low-level parallel libraries such as MPI, OpenMP, and UPC require advanced programming knowledge and sophisticated control of data transfer between processors. To address the difficulty of low-level schemes, higher-level libraries have emerged that simplify concurrent programming by hiding many of the low-level nuts and bolts. Examples include Cilk++ for C++, Grand Central Dispatch for Objective C, the Parallel Computing Toolbox for Matlab, and the concurrency libraries for Java. While these libraries do not eliminate all of the perils of concurrent programming, they do allow the programmer to focus more on big picture algorithm issues rather than the minute details of message passing schemes.

This work presents a general framework for using one such library, the Java fork/join library, for efficiently solving multi-objective network optimization problems in modern multi-core computers. Java is the only high-level language to offer a structured fork/join library optimized for divide-and-conquer algorithms, and is thus particularly suitable for the Non-Inferior Set Estimation (NISE) approach that is efficient at calculating the supported solutions of a multi-objective problem (Cohon *et al.* 1979). Section 0 introduces the problem and defines variables used in later pseudocode. Section 0 begins with a description of the serial Non-Inferior Set Estimation (NISE) algorithm (Cohon *et*

al. 1979) for computing supported multi-criteria solutions, and then expands this to a proposed parallel implementation, called pNISE. Section IV presents a case study using pNISE for solving a biobjective shortest path problem on a large raster GIS network. Section V discusses some computational case study of this application to a biobjective shortest path problem. Section VI presents some improvements to pNISE for instances with large number of processors. Finally, section 0 provides conclusions and enumerates other problems where this approach could be beneficial.

II. Background

Multiobjective optimization involves the task of determining noninferior solutions when considering multiple conflicting objectives, and is inherently more complicated than a problem's single-objective counterpart due to the added objective dimensionality. Most of past work in multi-objective modeling is first described for the use of two objectives, as this is usually the simplest case. Accommodating three or more objectives necessitates more complicated bookkeeping than what is required for two objectives, as some facets of the intersecting neighboring solutions in three or higher dimensions may lie in the interior rather than on the boundary of the convex polytope (Solanki 1986). Aside from this issue however, the fundamental theorems used to solve for tradeoffs in two objectives can be relatively easily expanded to three or more objectives. For this reason, most of the literature is concerned with the resolution of biobjective problems. This paper takes this same approach and restricts the discussion to biobjective problems as well. Further discussion of the nuances and approaches for problems with more than two objectives can be found in Przybylski *et al.* (2010).

In 1979, three different papers appeared in the published literature that addressed the problem of finding efficient solutions to biobjective optimization problems. (Dial 1979) developed a process that involved finding up to a pre-specified number of supported points to a biobjective shortest path problem, Aneja and Nair (1979) developed an approach to find all supported points to a biobjective transportation problem, and Cohon *et al.* (1979) developed a process to find non-dominated solutions to biobjective linear programming problems. Overall, all three techniques are quite similar, but do differ in their main focus. For example, Dial's approach runs until it finds a certain number of solutions or finds the complete tradeoff curve. The choice of problems solved, and hence the resolved tradeoff curve is based upon a recursion formula taking problems in order. Aneja and Nair's approach is similar to that of Dial's except it does not stop until it has resolved all parts of the tradeoff curve. Cohon et al. (1979) show how lower and upper bounds on the tradeoff curve can be defined as supported points are added to the tradeoff curve. This allows one the opportunity to resolve at each iteration that portion of the curve with the greatest estimation error. This technique is called the Non-Inferior Solution Estimation (NISE) technique. The NISE technique will either generate all supported points on a tradeoff curve within a set estimation bound limit, or can be executed to completion to generate all supporting points as suggested by Aneja and Nair. In this paper, we adopt the NISE method of Cohon *et al.* as it can be considered the most general of the three techniques. NISE (also known by various other names) has become the standard method in the literature for solving the supported solutions of a multiobjective problem, due to its efficiency and applicability with a wide range of solver techniques (Current *et al.* 1990, Ehrgott and Wiecek 2005, Daskalakis *et al.* 2010, Clímaco and Pascoal 2012). It is used as part of the preferred approach in a wide range of multiobjective applications, including forestry and agriculture (Fischer and Church 2003, Pyke and Fischer 2005, Kasprzyk *et al.* 2009, Breschan and Heinimann 2013), transmission and power flow systems (Salgado and Rangel Jr 2012, Soliman and Mantawy 2012, Medrano and Church 2014), industrial operations and logistics (Schilling

1982, Weber and Ellram 1993, Reklaitis 1996), and medical operations (Medaglia *et al.* 2009), just to name a few.

While NISE was originally presented within the context of biobjective linear programming problem, it can be applied to finding the supported solutions to biobjective Integer Programming (IP) or Mixed Integer Programming (MIP) problems as well. Modern first-rate MIP solvers have parallelism built-in to take advantage of multicore architectures; but specialized network optimization algorithms can often solve graph problems more efficiently than a general MIP solver. Tarapata (2007) published a comparison between solving a multiobjective shortest path problem on CPLEX vs. using a Dijkstra solver, and found that on large problems Dijkstra's computation times were 70 to 80 times faster than CPLEX.

IP problems can also have non-convex, non-inferior solutions known as *unsupported* solutions. Unsupported solutions are much more difficult to compute, as solving for those is equivalent to adding a knapsack constraint to the problem, which has been proven to be NP-hard (Garey and Johnson 1979). Some work has been published by Sanders and Mandow (2013) on a parallel biobjective shortest path algorithm for unsupported solutions, but this method introduces complicated and expensive data structures and introduce significant computational overhead in comparison to the fastest serial methods. A more recent method has been published that tries to reduce this overhead (Erb *et al.* 2014), although it is difficult to judge its effectiveness since the publication lacks any comparison with the fastest serial methods. This report focuses on finding only the supported solutions of either an LP or IP problem in parallel.

The methods described in this paper are applicable to a variety of specialized network algorithms, including but not limited to biobjective variants of the minimum spanning tree problem, classical transportation problem, assignment problem, maximum flow problem, and the minimum cost flow problem. This work has chosen to apply the NISE approach though to a biobjective shortest path problem using a form of Dijkstra's shortest path algorithm (Dijkstra 1959) with a binary heap priority queue (Cherkassky *et al.* 1996) as the optimization solver. As a point of reference, we compared computation times of our Dijkstra solver implementation to the native Matlab version on a single-objective problem. The Matlab function is called `graphshortestpath()`, and also uses a binary heap priority queue. When solved on various problems on two different 1000x1000 raster network data sets, the Matlab runtimes were consistently at least 2.2x longer than our Java version.

The biobjective shortest path problem is defined as follows. Let G = (N, A) be a directed graph network with node set $N = \{u_1, u_2, \ldots, u_n\}$ and arc set $A = \{(u_1, v_1), \ldots, (u_m, v_m)\}$. Each arc $(u,v) \in A$ has associated with it two positive real costs $c_{uv} = (c^1_{uv}, c^2_{uv})$. The biobjective shortest path problem aims to solve for the minimum-cost paths from a source node $s \in N$ to a destination node $t \in N$ that minimizes two, often competing, objectives, z_1 and z_2. Each arc has associated with it a decision variable x_{uv} that is equal to 1 if it lies on the optimal shortest path, and 0 otherwise. This results in the following problem formulation:

$$\min \; z_1(x) = \sum_{(u,v)\in A} c^1_{uv} x_{uv}$$

$$\min \; z_2(x) = \sum_{(u,v)\in A} c^2_{uv} x_{uv}$$

$$\text{s.t.} \quad \sum_{(u,v)\in A} x_{vu} - \sum_{(v,u)\in A} x_{vu} = \begin{cases} 1 & \text{if } u = s \\ -1 & \text{if } u = t \\ 0 & \text{if } u \neq s,t \end{cases} \qquad (1)$$

$$x_{uv} = \{0,1\} \text{ for all } (u,v) \in A$$

While the above formulation contains two distinct objectives, supported solutions may be found by solving the weighted combined single-objective formulation, using the weight α, where $0 \leq \alpha \leq 1$.

$$\min z_C(x) \; = \; \alpha \times z_1(x) \; + \; (1-\alpha) \times z_2(x) \qquad (2)$$

Different supported solutions may be computed by varying the weight between the two objectives. Setting $\alpha = 1$ finds the optimal solution considering only the first objective, while setting $\alpha = 0$ finds the optimal solution with respect to the second objective, and setting α to something in between to find compromise solutions on the trade-off curve. While it is possible to find a number of supported solutions by iteratively stepping the weight value, the NISE method (described in the next section) specifies a procedure to find all distinct supported solutions with a minimum number of total solver iterations, or to solve for a set of supported points and stop when all points within an estimation bound have been defined.

Each solution to the combined objective of equation generates an *s*-*t* path that is a supported non-dominated solution, i.e. σ_i is an optimal solution for a given α. Additionally, let $x_{uv}(\sigma_i)$ be the value of the variable x_{uv} in the σ_i solution, where the value is 1 if arc (u, v) is on the shortest path, and 0 otherwise. For a given path solution σ_i, the $z_1(\sigma_i)$ is its objective value with respect to the first objective, and $z_2(\sigma_i)$ is its objective value with respect to the second objective, as defined below. The term $z_C(\sigma_i, \alpha)$ represents the combined weighted objective according to the weight α.

$$z_1(\sigma_i) = \sum_{(u,v)\in A} c^1_{uv} x_{uv}(\sigma_i) \qquad (3)$$

$$z_2(\sigma_i) = \sum_{(u,v)\in A} c^2_{uv} x_{uv}(\sigma_i) \qquad (4)$$

$$z_C(\sigma_i, \alpha) = \alpha \times z_1(\sigma_i) + (1-\alpha) \times z_1(\sigma_i) \qquad (5)$$

The set $\Psi = \{\sigma_1, \sigma_2, \ldots, \sigma_p\}$ is the set of all supported non-dominated solutions to the problem, and form a convex Pareto frontier when plotted in objective space.

III. Supported Solution Search

A. Serial Non-Inferior Set Estimation (NISE)

The NISE method is used to find a set or subset of noninferior solutions of a biobjective linear, integer, or mixed-integer programming problem. Here, we describe this method to find all supported points of a trade-off curve. NISE begins by initially computing the single-objective solutions for each objective. In the biobjective case, these involve using weights $\alpha = 0$ and $\alpha = 1$. Once these solutions are determined, a weighting is chosen with equation 6 such that the z_c value for the two solutions are equal

$$\alpha = \frac{\left(z_2(\sigma_i) - z_2(\sigma_j)\right)}{\left(z_1(\sigma_i) - z_1(\sigma_j)\right) + \left(z_2(\sigma_i) - z_2(\sigma_j)\right)} \qquad (6)$$

Figure 1 graphically depicts how the selection of α creates an objective line where the two initial solutions, σ_1 and σ_2, have equal combined objective values. With this weighting, the problem can be solved again to find a solution that minimizes this weighted combined objective, denoted by σ_3.

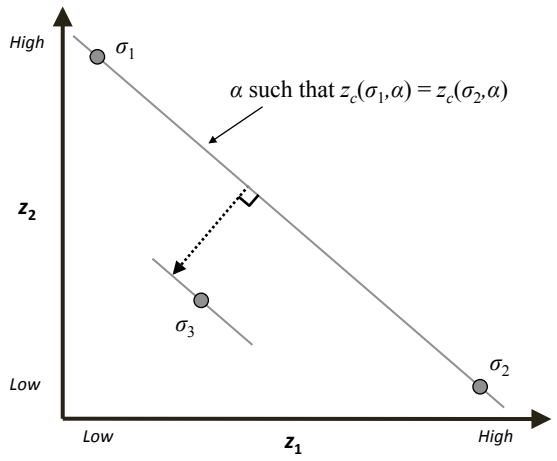


Figure 1. Objective space: σ3 solves min zc(x) with weight α

After solving for σ_3, new weightings can be determined to find solutions that minimize the combined objective between the new adjacent supported points. Figure 2 shows a new objective line to find a solution σ_4 between σ_1 and σ_3, and another objective line for finding a solution σ_5 between σ_3 and σ_2. If a combined objective returns a solution that does not improve the combined objective from the previously found solutions, then there are no supported points that expand the convex hull between those respective solutions and the search in that region is terminated. This process continues until all adjacent points have not had any new solutions found between them, and thus all supported solutions have been found.

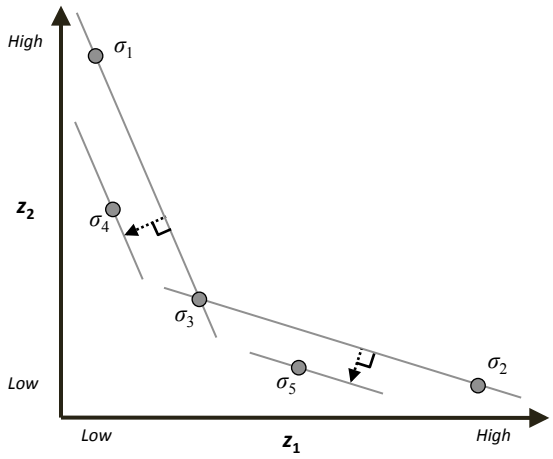


Figure 2. Objective space: supported solutions between σ_1 and σ_3, and between σ_3 and σ_2

Overall, NISE is a divide-and-conquer approach, and the general algorithm can be represented compactly with recursive function calls. The following pseudocode uses the NISE method for solving a biobjective shortest path problem using an optimal shortest path solver. The solver used in this work was Dijkstra's Algorithm with a binary heap priority queue (Cherkassky *et al.* 1996), although other specialized network algorithms could be used instead. In addition to the minimization problem presented, the code applies equally to a maximization problem by reversing the inequality in the dominance check.

Preliminary Algorithm: NISE for Biobjective Shortest Paths

```
// z_c(x) = α*z_1(x) + (1— α)*z_2(x)
// Ψ = {σ_1, σ_2,…, σ_p} = the set of supported non-dominated solutions
// Dij(α) solves a shortest s-t path with Dijkstra's algorithm using a
// combined objective weighted by α
// SetA(σ_i, σ_j) selects next value of α based on the z_1 and z_2 values of
// σ_i and σ_j
// RecursiveNISE(σ_i, σ_j) computes a supported solution between σ_i and σ_j

function: main
α = 1                          // minimize first objective
σ_i = Dij(α)
α = 0                          // minimize second objective
σ_j = Dij(α)
Ψ = σ_i
Ψ += RecursiveNISE(σ_i, σ_j)     // begin recursive NISE procedure

function: RecursiveNISE(σ_i, σ_j)
α = SetA(σ_i, σ_j)               // calculate alpha weighting, equation 6
σ_k = Dij(α)                     // solve composite objective
if (z_c(σ_k, α) < z_c(σ_i, α))          // if soln improves the composite
objective
Ψ += RecursiveNISE(σ_i, σ_k)
Ψ += RecursiveNISE(σ_k, σ_j)
else
Ψ += σ_j                         // else if no improvement found, return σ_j
end
return Ψ
```

The above algorithm though is a simplified version, and does not account for various anomalies that may occasionally arise. The next section lists these anomalies and how to deal with them, followed by a more comprehensive pseudocode that accounts for these scenarios.

B. NISE Anomalies

There are a few situations where one must take care in implementing the NISE method to avoid false-positive solutions or a non-terminating recursion causing a stack overflow exception. The following details these possible pitfalls, and how to avoid them.

1. Weakly Dominated Single Objective Solutions

The initial stage of the method requires solving the problem for each single objective. Oftentimes, there may exist numerous solutions that equally optimize that one objective. With regard to that objective, any of those solutions is optimal, yet they may perform quite differently from one-another when considering the other objectives in the model. In fact, in the initial single-objective base cases, an optimal solution may be returned that is weakly dominated by other equally optimal solutions. Such a solution is considered inferior, and should be omitted from the final non-dominated solution set.

For example, suppose one is minimizing $z_1(x)$ in the initial base case, as shown in Figure 3. The solver may return the solution σ_1, which is a minimum feasible solution to the problem with respect to objective 1. But there may exist another solution that was not found by the solver, σ_1', that weakly dominates σ_1, i.e. $z_1(\sigma_1) = z_1(\sigma_1')$ and $z_2(\sigma_1) > z_2(\sigma_1')$.

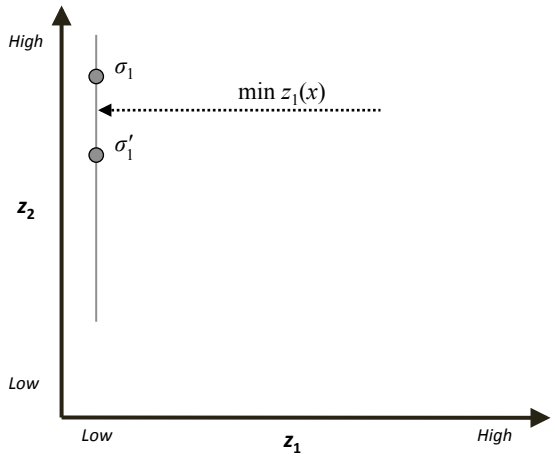


Figure 3. Weakly dominated solution that minimizes $z_1(x)$

Later in the algorithm, σ_1' will be found as the solution to a combined objective where α is very close to 1. A proper algorithm will put in place mechanisms to detect that it dominates σ_1 in order to eliminate it from the final solution.

2. Multiple Equal Value Composite Solutions

Another anomaly arises when solving a composite objective function, i.e. $0 < \alpha < 1$, where there are numerous solutions with the same composite objective value. Figure 4 shows what this scenario would look like when plotting the solutions in objective space. In this case, for a given α, $z_C(\sigma_i, \alpha) = z_C(\sigma_j, \alpha) = z_C(\sigma_k, \alpha)$. If σ_i and σ_k were the points used to determine α, and the solution returned is σ_j, then there is no problem. σ_j is a non-dominated solution that is on the convex Pareto-frontier. While its presence does not change the shape of the convex region, i.e. it is not a corner point; it is an optimal trade-off solution that should be kept. The NISE solution approach does not guarantee finding all solutions that are not corner points, but some may be found by chance.

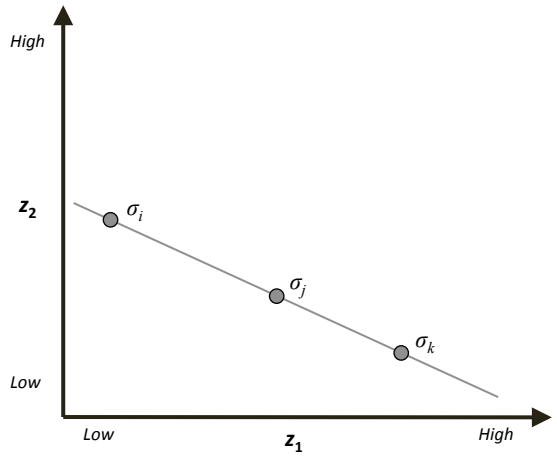


Figure 4. Multiple composite objective optimal solutions

The problem arises when σ_i and σ_j are the "outer points", i.e. `RecursiveNISE(σ_i, σ_j)`, and the solution returned is σ_k. If that point is kept, then the algorithm splits and runs `RecursiveNISE(σ_i, σ_k) and RecursiveNISE(σ_k, σ_j).` If `RecursiveNISE(σ_i, σ_k)` returns σ_j, then there is a situation of an endless cycle alternating between those solutions. With a recursive function, this will result in a stack overflow error, as the function will continue recursing ad infinitum until memory runs out.

In order to prevent this error and also to keep non-dominated solutions that are not corner points, rather than checking if an improvement is made to the combined objective z_c, a different criterion should be used to control if the function should recursively split. The alternative is to check if the returned solution is lexicographically in-between the outer points. If it is, then keep and split. Otherwise, the solution is lexicographically outside of the points, and the recursion ends and returns the appropriate solution. The next section revises the previous NISE pseudocode to take into account these two anomaly situations.

C. Complete NISE Pseudocode

Complete Algorithm: NISE for Biobjective Shortest Paths

```
// z_c(x) = α*z_1(x) + (1— α)*z_2(x)
// Ψ = {σ_1, σ_2,…, σ_p} = the set of supported non-dominated solutions
// Dij(α) solves a shortest s-t path with Dijkstra's algorithm using a
// combined objective weighted by α
// SetA(σ_i, σ_j) selects next value of α based on the z_1 and z_2 values of
// σ_i and σ_j
// RecursiveNISE(σ_i, σ_j) computes a supported solution between σ_i and σ_j

function: main
α = 1                          // minimize first objective
σ_i = Dij(α)
α = 0                          // minimize second objective
σ_j = Dij(α)
Ψ = σ_i
Ψ += RecursiveNISE(σ_i, σ_j)   // begin recursive NISE procedure
if (z_1(σ_1) == z_1(σ_2))      // if σ_2 dominates σ_1
Σ.removeFirstElement()
end

function: RecursiveNISE(σ_i, σ_j)
α = SetA(σ_i, σ_j)             // calculate alpha weighting, equation 6
σ_k = Dij(α)                   // solve composite objective
// if σ_k is lexicographically between σ_i and σ_j
if ((z_2(σ_k) < z_2(σ_i)) and (z_1(σ_k) < z_1(σ_j)))
if (z_2(σ_k) == z_2(σ_j))      // if σ_k weakly dominates σ_j
Ψ += RecursiveNISE(σ_i, σ_k)
else if (z_1(σ_k) == z_1(σ_i)) // if σ_k weakly dominates σ_i
Ψ += σ_k
Ψ += RecursiveNISE(σ_k, σ_j)
else                    // else σ_k is non-dominated
Ψ += RecursiveNISE(σ_i, σ_k)
Ψ += RecursiveNISE(σ_k, σ_j)
end
else
Ψ += σ_j                       // else if no improvement found, return σ_j
end
return Ψ
```

D. Java Fork/Join Framework

Java is a cross-platform object-oriented programming language that is ubiquitous in scientific computing, as well as in general desktop and mobile computing. It was originally released by Sun Microsystems in 1995, and is currently owned and actively developed by Oracle Corporation. One of the areas of Java language development since 2000 has been in its concurrency libraries. In September 2004, Java 5 was released which for the first time included the java.util.concurrent application programming interface (API) that included various low-level tools for simultaneously processing numerous threads. Developers saw a further need for higher-level concurrency tools that were implicitly scalable over a wide variety of hardware configurations, and the fork/join

framework was introduced by Doug Lea to address this need though the Java Community Process as a Java Specification Request, JSR 166 (Lea 2000, Lea 2003, Lea et al. 2004).

Fork/join is specifically designed to handle the difficult task of adding concurrency to recursive divide-and-conquer methods. Concurrent divide-and-conquer methods solve a problem by recursively splitting them into small subtasks, that are then solved in parallel, waiting for them to complete, and then composing results into a final answer. This approach is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g. quicksort, merge sort), multiplying large numbers, syntactic analysis (e.g. top-down parsers), convolution filters for digital image processing, and computing discrete Fourier transform (FFTs). The NISE algorithm described in this paper, used for determining the supported solutions to a biobjective optimization problem, also follows this general design paradigm.

The work breakdown of a divide-and-conquer algorithm tends to take a tree structure, where the task is split numerous times until a stopping criterion is reached, as shown graphically in Figure 5. For sorting or image processing, the stopping criteria may be dividing the problem into adequately small sub-problems; or in the case of NISE, the division stops for a specific region of the trade-off curve when no new supported solution is found in between two others. At this point, the results of the computation are sent back up the tree hierarchy, implicitly retaining the organized structure of the division, until all results have reached the top level and the final result is complete. Fork/join task trees may be symmetrical, as is typically the case for most divide-and-conquer algorithms, but may also be asymmetrical, as is the case with NISE.

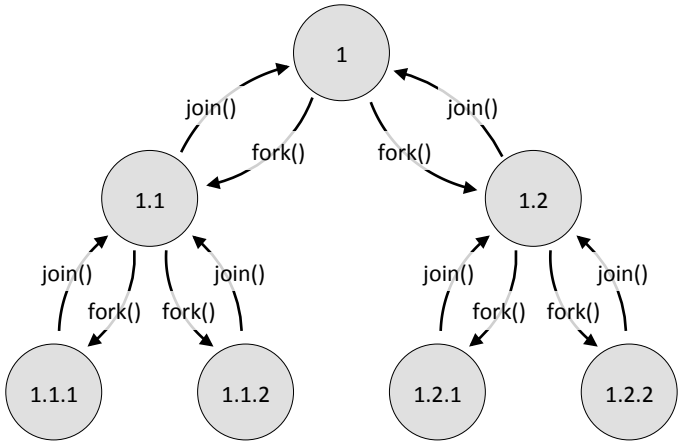


Figure 5. Fork/join task division

The Java implementation of fork/join uses a `ForkJoinPool` executor to manage the asynchronous concurrent execution of tasks. Tasks to be managed by the `ForkJoinPool` must implement the `ForkJoinTask` interface. `ForkJoinTask` objects feature two methods for performing their function: the `fork()` method launches a new task as a subtask of the one that called it, allowing it to be executed asynchronously; and, the `join()` method returns the results to the higher level task. A task cannot be joined until

all of its sub-tasks have joined into it, ensuring that all computations are completed before going back up the hierarchy. The Java implementation of `ForkJoinPool` is capable of "work stealing", which actively steals and reallocates tasks when a processor is waiting for a sub-task to complete and there are other pending tasks remaining to be computed. This helps to ensure balanced workloads across processors, improving the overall parallel efficiency of the application.

IV. Parallel NISE (pNISE)

A. Parallel Divide and Conquer

The general usage of the fork/join design pattern takes the following form:

```
if (my portion of the work is small enough)
   do the work directly
else
   split my work into two pieces
   invoke the two pieces and wait for the results
end
```

For the purposes of NISE though, it is necessary to first run a solver iteration in order to determine whether to divide the problem once more. To accomplish this, the following modification to the design pattern is used:

```
optimize weighted composite objective
if (the problem is indivisible)
   return the result
else
   split problem into two sub-problems
   invoke the two sub-problems and wait for the results
   return list of results
end
```

Finer nuances are necessary for handling if a weakly dominated extreme point is detected, in which case then the program needs to create a single sub-problem without a split.

B. Parallel Single-Objective Extreme Points

In addition to the binary tree generated from the recursive task division, the initial base case of the NISE algorithm requires two independent runs (in the biobjective case) of a network optimization solver. These can also be set up to be run in parallel, and since this is a general iterative procedure (rather than recursive), the simplest way of doing so is with multithreading using Java's `Thread` object. In this case, the solver is initialized within two independent threads, run simultaneously, and the `join()` method of `Thread` is used to wait until both threads have completed before proceeding with the remainder of the program.

If desired, one could avoid threads altogether, and continue using fork/join for the two base cases. While fork/join is intended for use on recursive functions, one can trick it for use on an iterative function by creating a wrapper class. Below is a pseudocode generalization of how this wrapper class is structured, called `SolverWrapper`. It takes two arguments: the first is a control boolean, and the second is the α value. One begins by calling `SolverWrapper(true,-1)`, where in this case the second argument is redundant and can take on any value. With an initial control argument of `true`, the program proceeds to split into two sub-problems, which are solved simultaneously using the

fork/join functionality. Each sub-problem is given a control argument of `false` and α values of 0 and 1 respectively, corresponding to the two single-objective optimizations.

```
class SolverWrapper(boolean toggle, double a)
if (toggle == false)
   Solve(a)       // optimize with composite weight a
   return result
else
   SolverWrapper(false,0)
   SolverWrapper(false,1)
   return list of results
end
```

Experiments indicated that no significant parallel performance difference between using threads or a wrapper fork/join class for the initial base cases, possibly due to the fact that only two problems were being solved.

V. Computational Case Study

A. Test Networks

While applicable to numerous multi-criteria network problems, the motivation behind this work was to develop tools to better enable the generation of noninferior alternatives to a transmission line corridor location problem. Thus, the performance of the pNISE procedure was evaluated by running a biobjective shortest path analysis on a GIS-based raster data set assembled and used by the Eastern Interconnection States' Planning Council (EISPC). This data set is intended to facilitate the identification of potential energy sites and transmission line corridors within the EISPC region, which spans 39 eastern U.S. states, Washington D.C., and 8 Canadian provinces. The data was assembled jointly by Argonne National Laboratory, Oak Ridge National Laboratory, and the National Renewable Energy Laboratory as a part of their EISPC's Energy Zones Study (EZS) (Kuiper *et al.* 2013).

The EZS data contains numerous geographical information layers that would be used in a suitability analysis for locating new energy infrastructure, and is available through the EISPC Energy Zones Mapping Tool (EZMT, eispctools.anl.gov). The EZS includes 250 data layers, including such things as land cover type, slope, water bodies, watersheds, essential habitats, earthquake intensities, existing transmission lines, substations, rail and roadways, just to name a few. This work used a 1000x1000 raster subset of the EZS data, with a 250 square meter cell size. The region analyzed was in the Kentucky Lake region where the Tennessee River and the Cumberland River intersect the Ohio River; and includes portions of Tennessee, Kentucky, Illinois, and Missouri.

The case study involved the slope and land cover type layers for the two objectives, as these roughly correspond to the competing objectives of cost vs. environmental impact respectively. Slope values were in percent slope, and land cover was already categorized according to the National Land Cover Database 2006 (Fry *et al.* 2011). These values and categories were converted to cell costs according to the terrain cost multipliers recommended by the Western Electricity Coordinating Council (Mason *et al.* 2012). Figure 6 displays graphics of the EISPC data maps used in the analysis, represented as 1000x1000 rasters and classified with high costs in dark colors and low costs in light colors. The left map represents the environmental impact objective, and the right map represents the construction cost objective.

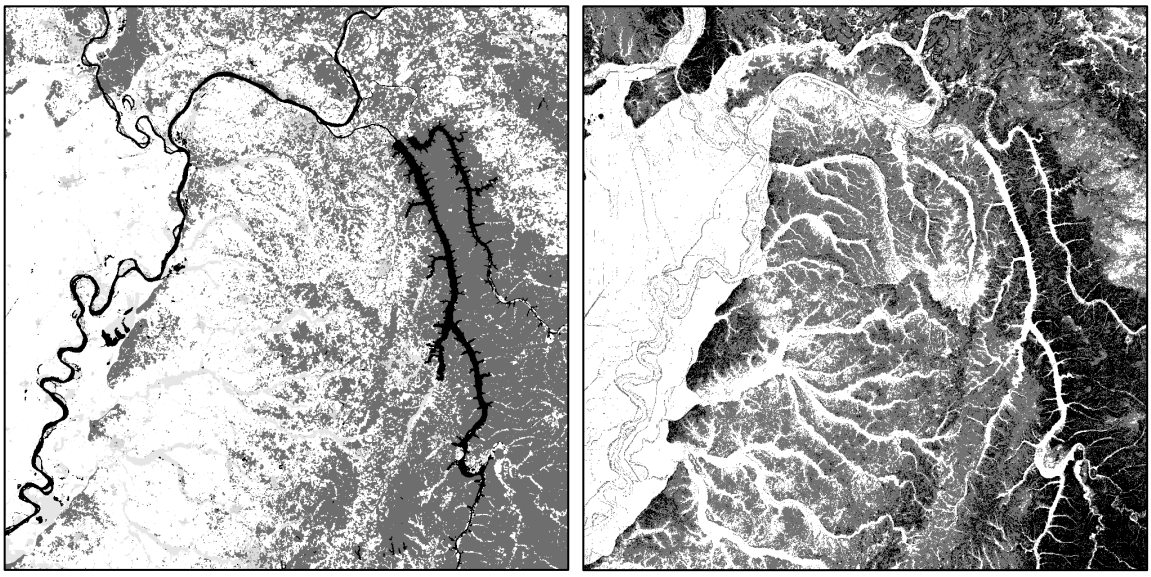


Figure 6. EISPC maps classified into two objectives: environmental impact (left), and construction cost (right)

From the raster layers, networks were created according to the guidelines of Huber and Church (1985), whereby nearby raster nodes were connected with arcs, and the arc cost labels for each objective assigned as a function of the node costs and the geometry of the arc itself. Three network versions were generated from the raster, with *r* radius values of 0, 1, and 2 respectively. The $r = 0$ network corresponds to an orthogonal grid, $r = 1$ adds diagonal "queen's moves", and $r = 2$ adds to that "rook's moves". Each higher value *r*-network decreases the inherent geometric distortion of routes at the expense of adding more arcs and thus increasing computation time. Higher order networks are possible, but the increase in computational effort is not justified due to diminishing returns in spatial accuracy. According to Huber and Church, "the second order system ($r = 2$) appears to provide the most satisfactory trade-off between accuracy and computational burden."

Experiments were run on the 1000x1000 network on four origin/destination (OD) pairs: OD1 was from the SW corner to the NE corner, and OD2 was from the NW corner to the SE corner. The other OD pairs that were tested used starting and ending points closer to one another. Table 1 lists the networks used, and their properties including the coordinates of the OD nodes, *r*-value, number of nodes and arcs, and the number of supported noninferior solutions for that problem. Cells of the raster are referenced by the rows and columns, with the top-left corner cell being referenced as (0, 0). Row numbers increase as one heads south, and column numbers increase as one goes heading east.

Table 1. EISPC Test Networks Properties

OD Name	Origin Node	Destination Node	r	Total Nodes	Total Arcs	Supported Noninferior Solutions
1	(999, 0)	(0, 999)	0	1,000,000	3,996,000	86
			1	1,000,000	7,988,004	138
			2	1,000,000	15,964,020	266
2	(0, 0)	(999, 999)	0	1,000,000	3,996,000	89
			1	1,000,000	7,988,004	153
			2	1,000,000	15,964,020	274
3	(699, 300)	(300, 699)	0	1,000,000	3,996,000	32
			1	1,000,000	7,988,004	69
			2	1,000,000	15,964,020	114
4	(599, 400)	(400, 599)	0	1,000,000	3,996,000	23
			1	1,000,000	7,988,004	39
			2	1,000,000	15,964,020	69

B. Experimental Procedures

In order to test the efficacy of the pNISE approach, simulations were run on different hardware running Java version 7u51. One of the greatest strengths of fork/join and Java in general is that it is cross-platform and automatically scalable, thus no modifications are necessary in order to run the code on different hardware. The first experiment compared the speedup of the pNISE versus an equivalent serial NISE implementation on a quad-core laptop running Apple OS X v10.9.2. The second experiment was a scaling experiment, evaluating the speedup and efficiency of pNISE based on the different numbers of allocated processors on a 32 core HP server running Red Hat Enterprise Linux Server release 6.2.

Metrics used to measure performenace included the speedup S_p and parallel efficiency E_p. Letting p be the number of processors, and T_p be the execution time of a parallel algorithm on p processors, then T_1 is the execution time for the serial (1-processor) version of the algorithm, and in the ideal scenario, $S_p = p$ and $E_p = 1$, although this rarely occurs in parallel computation applications except for trivially simple cases such as Monte-Carlo simulation. In addition to high speedup values, one also looks for a linear trend as the number of processors increases. This would indicate that a method is scalable to a very high number of processors while maintaining a good speedup. As with perfect speedup, linear speedup trends are typically not possible to maintain except in the case for very simple problems, since speedup is limited by the amount of parallelism that exists in a problem instance or program (Amdahl 1967). In the case of pNISE, the two initial base cases must be completed before commencing the recursive portion of the algorithm. While the base cases can compute in parallel, the maximum speedup is only 2 for that portion of the calculation, since only two threads exist. Even after the recursive portion begins, the task division progresses as a binary tree (Figure 5), starting with a single level of parallelism, followed by two, then four, and so on. For smaller problems, this time with less parallelism can take a significant amount of the total computation time, so less speedup will be expected. On the other hand, larger problems use a smaller

proportion of their total computation time in these inefficient phases, and thus would have a higher expected speedup.

C. Computational Results

1. Serial NISE vs. pNISE

The first experiment tested the serial implementation of NISE to the parallel pNise. The hardware used was an Apple computer with a 3.7 GHz Intel Core i7-3820QM quad-core processor and 16GB of RAM. Results from this analysis are summarized in

Table 2, comparing runtimes between a serial implementation of NISE using no concurrency, versus the fork/join pNISE approach. The results show that pNISE was able to maintain a high speedup in all cases, particularly for the largest problems (OD1 an OD2), which contain the most supported solutions. All OD1 and OD2 problems maintained speedup results between 3.32 and 3.56, with good mid-80% efficiencies. The smaller problems had a lower expected speedup due to a greater proportion of their total computation time being performed during the inefficient phases of the algorithm. This was evident with the OD3 problems having speedups of around 3.0 with mid-70% parallel efficiencies, and the smallest OD4 problems having 2.0-2.57 speedups and parallel efficiencies dropping to the 50%-65% range. In general, the larger the problem in terms of computation time and number of solutions, then the more efficient the parallelization.

Table 2. Serial NISE vs pNISE Runtimes and Speedup

OD	r	Supported Solns.	NISE T_1 (seconds)	pNISE T_4 (seconds)	S_4	E_4
1	0	86	117.490	34.394	3.416	0.854
	1	138	281.087	84.497	3.327	0.832
	2	266	950.571	267.178	3.558	0.889
2	0	89	117.319	35.245	3.329	0.832
	1	153	305.369	87.930	3.473	0.868
	2	274	981.298	281.162	3.490	0.873
3	0	32	29.748	10.019	2.969	0.742
	1	69	100.906	35.481	2.844	0.711
	2	114	285.074	95.004	3.001	0.750
4	0	23	10.484	4.075	2.573	0.643
	1	39	27.545	13.705	2.010	0.502
	2	69	87.031	36.333	2.395	0.599

2. pNISE Scaling

This experiment evaluated how the efficiency of the pNISE approach scales as the number of processors available is increased. `ForkJoinPool(p)` can take an optional input integer argument p that limits the executor service to that specified level of parallelism. All tests were run on a set of HP ProLiant DL580 Gen8 Server nodes, each with 512GB of RAM and four 2.0 GHz Intel Xeon X7550 8 core processors for a total of 32 cores per node. Results of this analysis are listed in Table 3. Only OD1 and OD2 were evaluated, as the other problems were too small to be able to fully make use of the large number of processors. Each of the cores on the nodes was approximately half as fast as a core on the

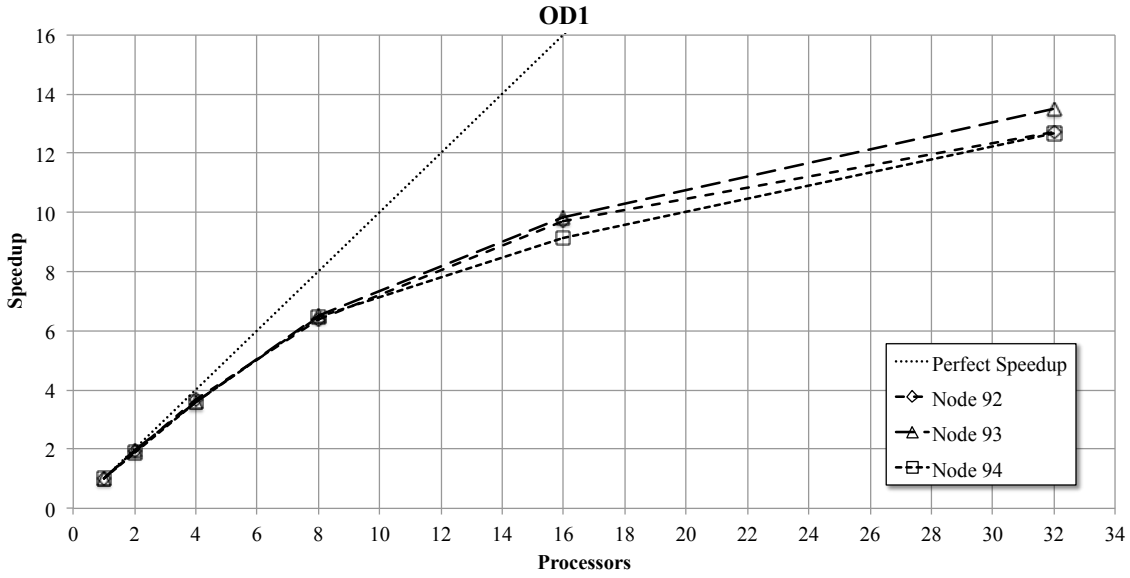
Apple computer, but higher speedups were achieved since there were eight times as many cores per machine.

Table 3. pNISE Scaling on 32 core server nodes

OD1 node 92				OD1 node 93				OD1 node 94			
p	T_p (seconds)	S_p	E_p	p	T_p (seconds)	S_p	E_p	p	T_p (seconds)	S_p	E_p
1	1896.593	1.00	1.00	1	1866.592	1.00	1.00	1	1816.335	1.00	1.00
2	974.751	1.95	0.97	2	945.792	1.97	0.99	2	956.488	1.90	0.95
4	512.794	3.70	0.92	4	522.349	3.57	0.89	4	508.790	3.57	0.89
8	297.818	6.37	0.80	8	286.318	6.52	0.81	8	281.113	6.46	0.81
16	195.762	9.69	0.61	16	190.039	9.82	0.61	16	199.161	9.12	0.57
32	148.945	12.73	0.40	32	138.434	13.48	0.42	32	143.380	12.67	0.40

OD2 node 92				OD2 node 93				OD2 node 94			
p	T_p (seconds)	S_p	E_p	p	T_p (seconds)	S_p	E_p	p	T_p (seconds)	S_p	E_p
1	2007.119	1.00	1.00	1	1999.339	1.00	1.00	1	1949.368	1.00	1.00
2	1029.290	1.95	0.98	2	1021.425	1.96	0.98	2	1019.560	1.91	0.96
4	552.009	3.64	0.91	4	520.650	3.84	0.96	4	517.533	3.77	0.94
8	316.857	6.33	0.79	8	304.344	6.57	0.82	8	295.011	6.61	0.83
16	200.411	10.02	0.63	16	201.775	9.91	0.62	16	198.758	9.81	0.61
32	150.484	13.34	0.42	32	146.503	13.65	0.43	32	147.584	13.21	0.41

In general, speedups increased but the parallel efficiency decreased as the number of processors used was increased. The reasons for the drop in efficiency as more cores are used is due to a combination of the program running a longer time with less parallelism than processors, as well as the effects of increased overhead in coordinating the larger executor pool. Even so, the parallelization achieved speedups of up to 13.65, significantly reducing the overall computation time for these large problems. The speedup performance and deviation from the theoretically ideal efficiency are evident in Figure 7, which is a plot of speedup vs. processors used. It should be noted that a problem involving additional objectives or larger number of supported points will likely result in higher speedups due to their larger problem size. Overall, the results reported here demonstrate the value and ease of parallelizing the NISE algorithm.



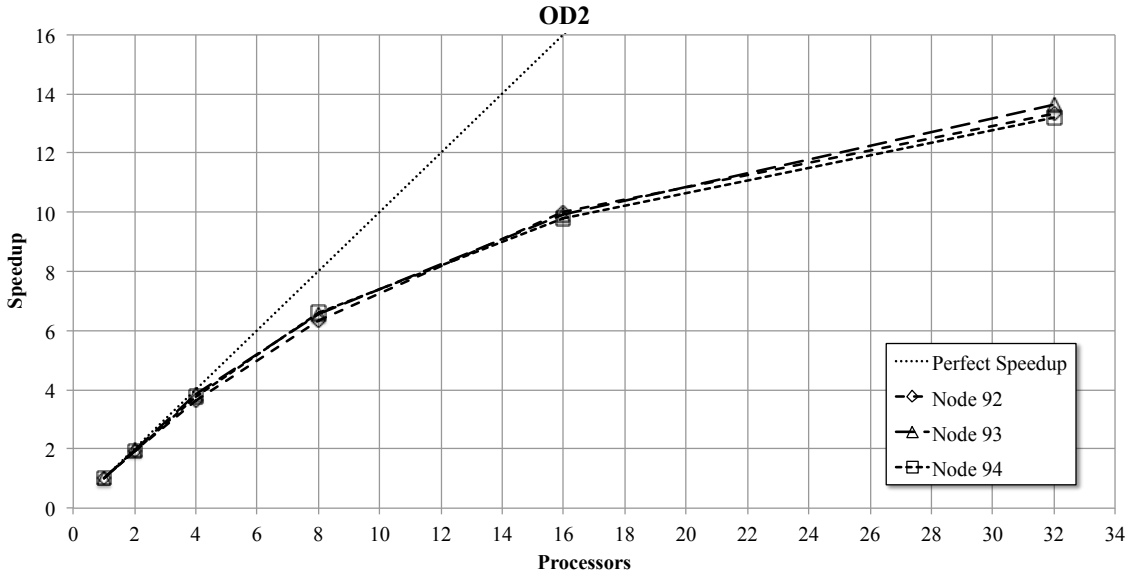


Figure 7. Scaling analysis of the speedup on 32 core nodes for OD1 (top) and OD2 (bottom)

VI. Improvements to pNISE Efficiency

A. Parallelism Analysis of Basic pNISE

The previous section detailed a method for using Java's Fork/Join concurrency to parallelize the search for supported solutions to a biobjective shortest path problem. While effective at producing 13x speedups on a 32 core processor machine, there is still room for improvement toward achieving a theoretically ideal 32x speedup. One issue is that the initial computation of the pNISE approach, where the weightings are 0 and 1, has only a level of parallelism of 2. When the solutions of these two are used to solve the next supported point, that has a parallelism of 1. Then next step then splits into two problems, with a parallelism of 2. Then 4, then 8, then 16, etc. After a few iterations, eventually there is more parallelism than there are processors and the method uses all resources efficiently from then on. But up until that point, some processors are waiting idly until enough parallelism exists to use all of the computational resources.

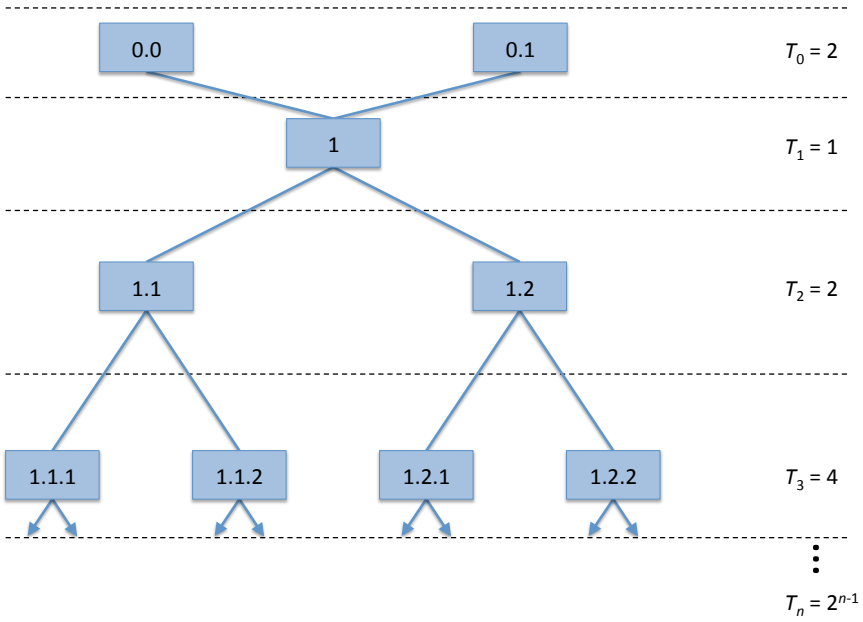


Figure 8. Parallel threads at each stage of pNISE

Figure 8 displays the parallelism at each stage in a graphical manner, showing the number of independent threads (i.e. the level of parallelism), T_i, at each stage i of the algorithm. pNise solves the weighted sum objective initially for 0 and 1 weights using two independent parallel threads. It then uses those solutions to determine the next weight, and solves a single problem. That solution is then used in conjunction with the two initial solutions to determine two more weights, and solve those. This process continues in a binary tree manner until all branches fathom with no further supported solutions to be found. Assuming no early fathoming, each stage n after the initial has a level of parallelism of $T_n = 2^{n-1}$. This means that for a 32 core computer, six stages of the

algorithm must be completed (including the initial zero stage) before there are enough threads in the thread pool to use all of the processors.

B. Improving Initial Parallelism

An improvement to the use of resources during the initial stages, rather than initially just solving the weighted objective for only values of 0 and 1, is to perform additional solver iterations on unused processors with weights in-between 0 and 1. This is done using simple threaded-parallelism, rather than a structured Fork/Join scheme. Essentially, given p processors, for each processor $i = 1 \ldots p$, execute the weighted sum objective for weight value of $z_i = (i\text{-}1) / (p\text{-}1)$ (i.e. equal intervals between 0 and 1). For a shortest path problem, these solutions take approximately the same amount of time to execute, and thus they all complete at approximately the same time. Once complete, a supported solution will have been found for each weighted graph, some of which may be repeated solutions, depending on the size of the problem and the number of processors. If there are no repeated solutions, then the p solutions can be used to initiate p-1 fork/join instances, which all get managed by the single Java thread pool in order to efficiently allocate work. From this point on, the procedure completes itself the same way as the original pNISE, in that a number of fork/join problems are solved in parallel, but the threads from all centrally handled by the Java threadpool.

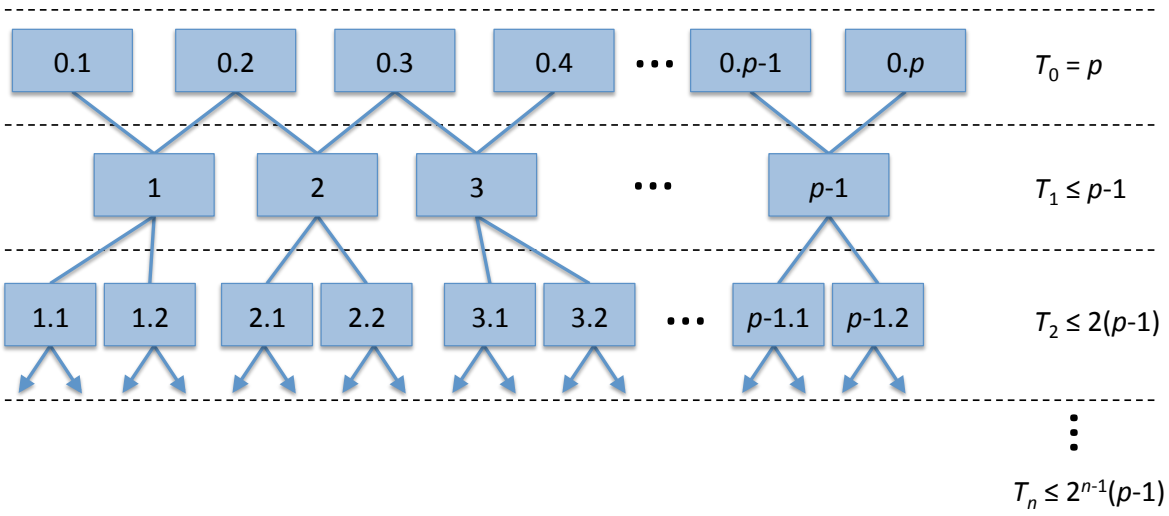


Figure 9. Parallel threads at each stage of the enhanced pNISE, making use of unused processors in the initial stages

Figure 9 displays this enhanced parallelism graphically. In the best-case scenario, the method uses all processors in the initial stage, all but one in the next stage, and all processors again for subsequent stages until computation is complete. This is in contrast with the original approach using 2 in the initial, then 1, then 2, then 4, then 8, etc. If there are repeated solutions in the initial stage, then there will be fewer than p-1 fork/join subproblems; but our experiments on the EISPC network never had fewer than 17 independent solutions in the initial stage, which means than in all cases all processors were always used by two stages later.

C. Computational Results

The same experiments were performed with the enhanced pNISE method, and compared with the simple pNISE approach. Table 4 shows the results of these computations for the two most-distant OD pairs: OD1 from the bottom-left corner to the top-right corner of the map, and OD2 from the top-left corner to the bottom-right corner on the map (see Figure 6). All computations were done on the number 92 node at UCSB's CSC for hardware consistency. On the left side of the table are computations using the simple pNISE, and on the right are using the enhanced pNISE. As the number of processors increase, the overall speedup and efficiencies improve, in the case of the 32 core experiment going from a speedup of 12.73 to a speedup of 15.32 for OD1, and going from a speedup of 13.24 to a speedup of 15.31 for OD2.

Table 4. Comparison of simple pNISE and enhanced pNISE on 32 core server nodes

Simple pNISE – OD1 node 92				Enhanced pNISE – OD1 node 92			
p	T_p (seconds)	S_p	E_p	p	T_p (seconds)	S_p	E_p
1	1896.593	1.00	1.00	1	1894.384	1.00	1.00
2	974.751	1.95	0.97	2	963.231	1.97	0.98
4	512.794	3.70	0.92	4	492.764	3.84	0.96
8	297.818	6.37	0.80	8	286.472	6.61	0.83
16	195.762	9.69	0.61	16	173.640	10.91	0.68
32	148.945	12.73	0.40	32	123.663	15.32	0.48

Simple pNISE – OD2 node 92				Simple pNISE – OD2 node 92			
p	T_p (seconds)	S_p	E_p	p	T_p (seconds)	S_p	E_p
1	2007.119	1.00	1.00	1	2004.572	1.00	1.00
2	1029.290	1.95	0.98	2	1007.387	1.99	0.99
4	552.009	3.64	0.91	4	519.601	3.86	0.96
8	316.857	6.33	0.79	8	282.882	7.09	0.89
16	200.411	10.02	0.63	16	177.483	11.29	0.71
32	150.484	13.34	0.42	32	130.963	15.31	0.48

These improvements are also evident when comparing the speedups graphically as depicted in Figure 10, where the benefits of the enhanced pNISE become more pronounced as the number of processors increase.

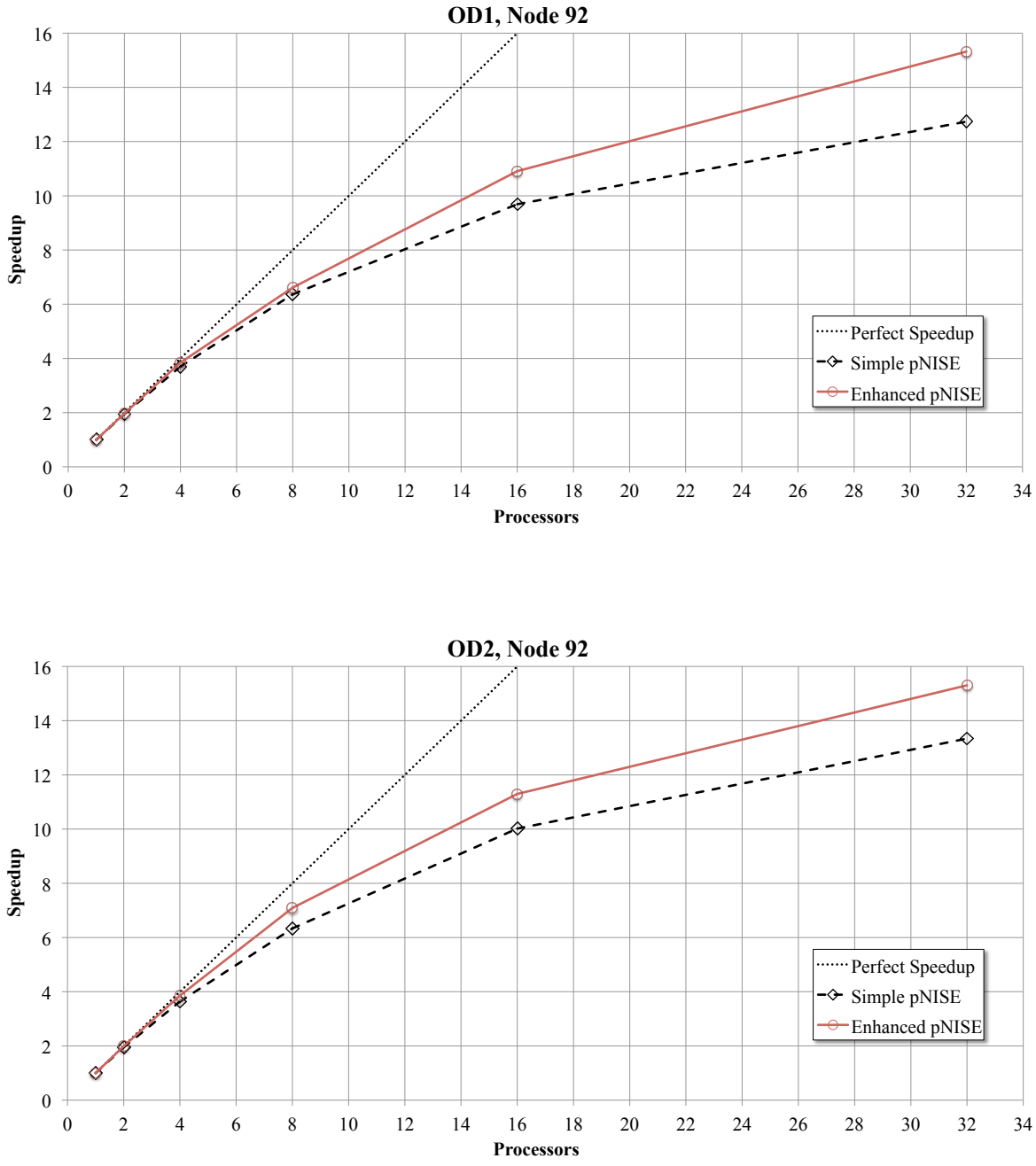


Figure 10. Speedup comparison between Simple pNISE and Enhanced pNISE for OD1 (top) and OD2 (bottom)

VII. Concluding Remarks

This work developed a "simple-to-program" parallel implementation of the NISE method for computing the supported non-dominated solutions to biobjective network optimization problems, called pNISE. This method uses high-level fork/join framework within the Java 7 concurrency API to make this method parallel without the difficult complexities of traditional low-level message-passing parallel languages. After describing how to develop this algorithm, a transmission line corridor location case study using a large real-world data set demonstrated that the pNISE was effective at taking advantage of modern multi-core computer processors to significantly reduce the computation time of the biobjective supported solution set. Additional enhancements to the pNISE approach were then described, which further improved the parallel performance of the method.

The pNISE approach is applicable to all network problems where there exist fast, specialized optimal solution algorithms. In addition to the biobjective shortest path problem evaluated in this paper (Raith and Ehrgott 2009, Medrano and Church 2014), other problems that could be solved with pNISE include biobjective variants of the minimum spanning tree problem (Steiner and Radzik 2008), classical transportation problem (Aneja and Nair 1979), assignment problem (Przybylski *et al.* 2008), maximum flow problem (Royset and Wood 2007), and minimum-cost flow problem (Hamacher *et al.* 2007), just to name a few. With the continuing expansion of big data, scientists and engineers must tackle larger network problems than ever before, requiring novel tools to enable multicriteria analysis and optimization on these massive data sets using modern computing resources. Using simple general-purpose parallel tools such as pNISE to speed up computation allows a designer to focus less time and energy on the generation of alternative solutions, and more time on model development and analysis to provide the best solutions to these challenging problems.

References

Amdahl, G.M., (1967). Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS*, Atlantic City, N.J.: ACM, 483-485.

Aneja, Y.P. & K.P.K. Nair, (1979). Bicriteria transportation problem. *Management Science,* 25, 73-78.

Breschan, J.R. & H.R. Heinimann, (2013). Ecoforest–automatic design of forest patterns attractive to wildlife in an artificial landscape. *Journal of Applied Operational Research,* 5, 125-134.

Cherkassky, B.V., A.V. Goldberg & T. Radzik, (1996). Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming,* 73, 129-174.

Clímaco, J.C.N. & M.M.B. Pascoal, (2012). Multicriteria path and tree problems: Discussion on exact algorithms and applications. *International Transactions in Operational Research,* 19, 63-98.

Cohon, J.L., R.L. Church & D.P. Sheer, (1979). Generating multiobjective trade-offs: An algorithm for bicriterion problems. *Water Resources Research,* 15, 1001-1010.

Current, J.R., C.S. Revelle & J.L. Cohon, (1990). An interactive approach to identify the best compromise solution for two objective shortest path problems. *Computers & Operations Research,* 17, 187-198.

Daskalakis, C., I. Diakonikolas & M. Yannakakis, (2010). How good is the chord algorithm? *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*Society for Industrial and Applied Mathematics, 978-991.

Dial, R.B., (1979). A model and algorithm for multicriteria route-mode choice. *Transportation Research Part B: Methodological,* 13, 311-316.

Dijkstra, E.W., (1959). A note on two problems in connexion with graphs. *Numerische Mathematik,* 1, 269-271.

Ehrgott, M. & M.M. Wiecek, (2005). Mutiobjective programming. *Multiple criteria decision analysis: State of the art surveys.* Springer, 667-708.

Erb, S., M. Kobitzsch & P. Sanders, (2014). Parallel bi-objective shortest paths using weight-balanced b-trees with bulk updates. *In* Gudmundsson, J. & Katajainen, J. eds. *Experimental algorithms.* Springer International Publishing, 111-122.

Fischer, D.T. & R.L. Church, (2003). Clustering and compactness in reserve site selection: An extension of the biodiversity management area selection model. *Forest Science,* 49, 555-565.

Fry, J.A., G. Xian, S. Jin, J.A. Dewitz, C.G. Homer, Y. Limin, C.A. Barnes, N.D. Herold & J.D. Wickham, (2011). Completion of the 2006 national land cover database for the conterminous united states. *Photogrammetric Engineering and Remote Sensing,* 77, 858-864.

Garey, M.R. & D.S. Johnson, (1979). *Computers and intractability,* Freeman San Francisco, CA.

Hamacher, H.W., C.R. Pedersen & S. Ruzika, (2007). Multiple objective minimum cost flow problems: A review. *European Journal of Operational Research,* 176, 1404-1422.

Huber, D.L. & R.L. Church, (1985). Transmission corridor location modeling. *Journal of Transportation Engineering-Asce,* 111, 114-130.

Kasprzyk, J.R., P.M. Reed, B.R. Kirsch & G.W. Characklis, (2009). Managing population and drought risks using many-objective water portfolio planning under uncertainty. *Water Resources Research,* 45.

Kuiper, J., D.P. Ames, D. Koehler, R. Lee & T. Quinby, (2013). *Web-based mapping applications for solar energy project planning.* Idaho National Laboratory, Preprint, INL/CON-13-28372.

Lea, D., (2000). A java fork/join framework. *Proceedings of the ACM 2000 conference on Java Grande*ACM, 36-43.

Lea, D., 2003. *Concurrency jsr-166 interest site* [online]. http://gee.cs.oswego.edu/dl/concurrency-interest/index.html.

Lea, D., J. Bowbeer, D. Holmes, B. Goetz & T. Peierls, 2004. *Jsr 166: Concurrency utilities* [online]. Java Community Process. Available from: http://www.jcp.org/en/jsr/detail?id=166.

Liebman, J.C., (1976). Some simple-minded observations on the role of optimization in public systems decision-making. *Interfaces,* 6, 102-108.

Mason, T., T. Curry & D. Wilson, (2012). *Capital costs for transmission and substations.* Black & Veatch prepared for WECC, Proj. No. 176322.

Medaglia, A.L., J.G. Villegas & D.M. Rodríguez-Coca, (2009). Hybrid biobjective evolutionary algorithms for the design of a hospital waste management network. *Journal of Heuristics,* 15, 153-176.

Medrano, F.A. & R.L. Church, (2014). Corridor location for infrastructure development: A fast bi-objective shortest path method for approximating the pareto frontier. *International Regional Science Review,* 37, 129-148.

Moore, G.E., 1965. Cramming more components onto integrated circuits. McGraw-Hill New York, NY, USA.

Przybylski, A., X. Gandibleux & M. Ehrgott, (2008). Two phase algorithms for the bi-objective assignment problem. *European Journal of Operational Research,* 185, 509-533.

Przybylski, A., X. Gandibleux & M. Ehrgott, (2010). A recursive algorithm for finding all nondominated extreme points in the outcome set of a multiobjective integer programme. *INFORMS Journal on Computing,* 22, 371-386.

Pyke, C.R. & D.T. Fischer, (2005). Selection of bioclimatically representative biological reserve systems under climate change. *Biological Conservation,* 121, 429-441.

Raith, A. & M. Ehrgott, (2009). A comparison of solution strategies for biobjective shortest path problems. *Computers & Operations Research,* 36, 1299-1331.

Reklaitis, G.V., (1996). Overview of scheduling and planning of batch process operations. *Batch processing systems engineering.* Springer, 660-705.

Royset, J.O. & R.K. Wood, (2007). Solving the bi-objective maximum-flow network-interdiction problem. *INFORMS Journal on Computing,* 19, 175-184.

Salgado, R. & E. Rangel Jr, (2012). Optimal power flow solutions through multi-objective programming. *Energy,* 42, 35-45.

Sanders, P. & L. Mandow, (2013). Parallel label-setting multi-objective shortest path search. *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*IEEE, 215-224.

Schilling, D.A., (1982). Strategic facility planning: The analysis of options. *Decision Sciences,* 13, 1-14.

Solanki, R.S., (1986). *Techniques for approximating the noninferior set in linear multiobjective programming problems with several objectives*. Ph.D. Johns Hopkins University.

Soliman, S.a.-H. & A.-a.H. Mantawy, (2012). Optimal power flow. *Modern optimization techniques with applications in electric power systems.* Springer, 281-346.

Steiner, S. & T. Radzik, (2008). Computing all efficient solutions of the biobjective minimum spanning tree problem. *Computers & Operations Research,* 35, 198-211.

Tarapata, Z., (2007). Selected multicriteria shortest path problems: An analysis of complexity, models and adaptation of standard algorithms. *International Journal of Applied Mathematics and Computer Science,* 17, 269-287.

Weber, C.A. & L.M. Ellram, (1993). Supplier selection using multi-objective programming: A decision support system approach. *International Journal of Physical Distribution & Logistics Management,* 23, 3-14.