

An *A*ynchronous and  
Fault Tolerant Algorithm for  
*P*arallel *P*attern *S*earch  
Optimization

TAMARA G. KOLDA

PATRICIA D. HOUGH

Sandia National Labs, Livermore

VIRGINIA TORCZON

College of William & Mary

# ACKNOWLEDGMENTS

DOE MICS Office  
Sandia National Labs  
National Science Foundation

# OUTLINE

- Pattern Search Methods for Optimization
- Parallel Pattern Search (PPS)
- Asynchronous Parallel Pattern Search (APPS)
- Example: A Thermal Design Problem
- Example: An Electrical Circuit Simulation
- Fault Tolerance in APPS
- Example: An Electrical Circuit Simulation *with Faults*
- Convergence Theory for APPS
- Conclusions & Future Work

# USING PATTERN SEARCH

Our goal is to solve the unconstrained optimization problem

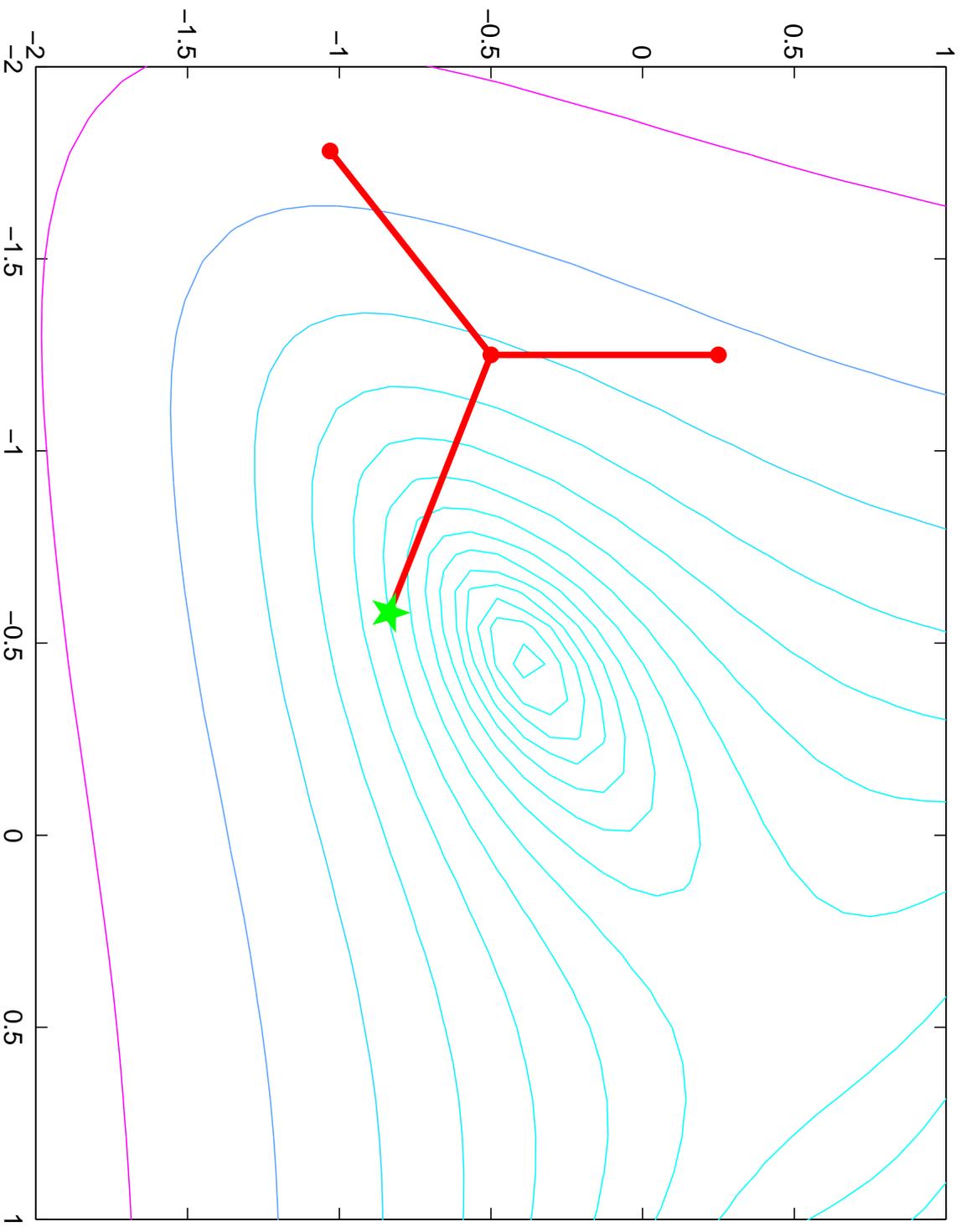
$$\min f(x) \quad x \in \mathcal{R}^n$$

without derivative information. Pattern search is a popular *derivative-free* method which is most useful when...

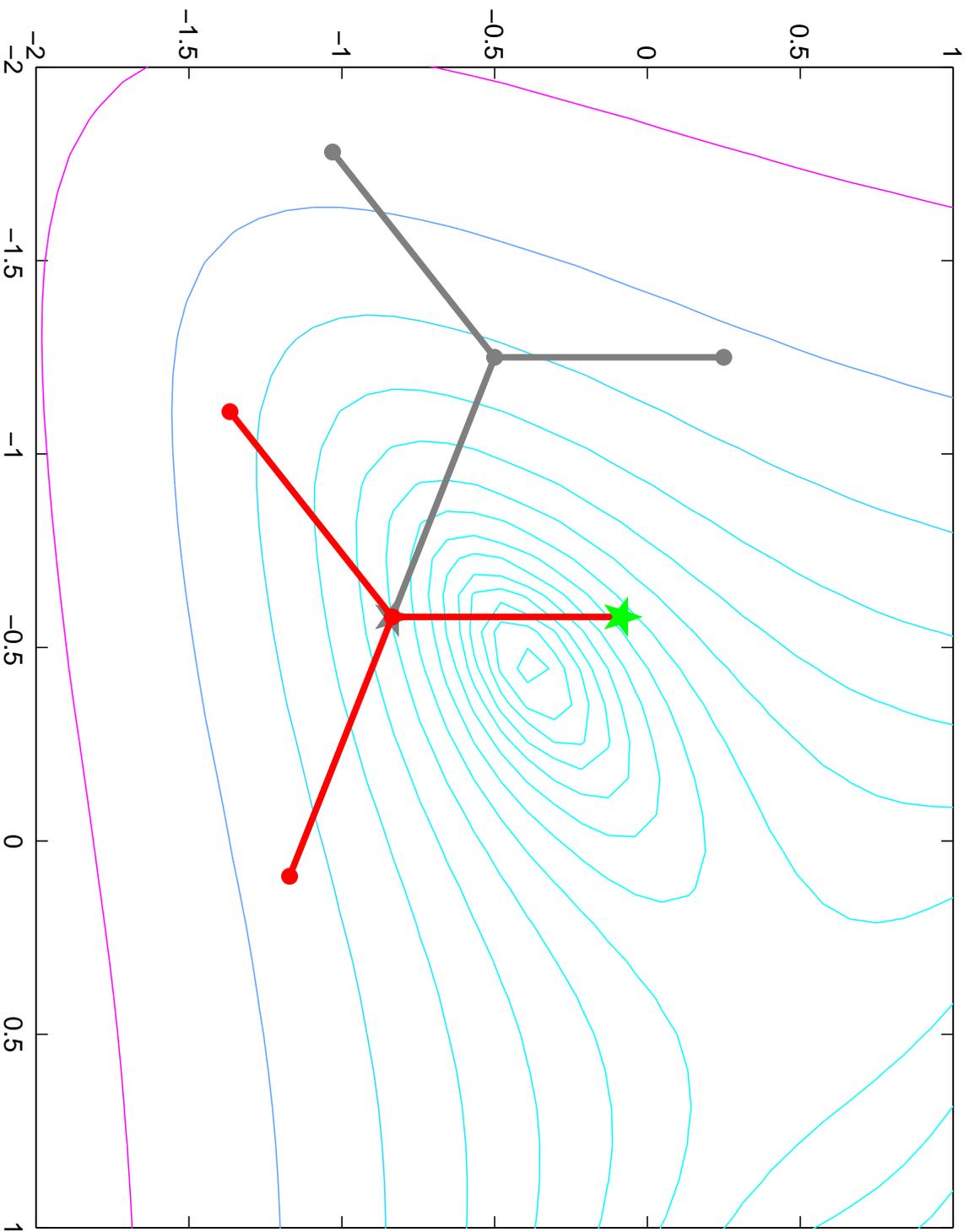
- The function is expensive to calculate.
- The gradient cannot be calculated.
- Numerical approximation of the gradient is too slow, or the function values are too noisy to yield reliable gradient approximations.

*Examples of such problems come from engineering applications.*

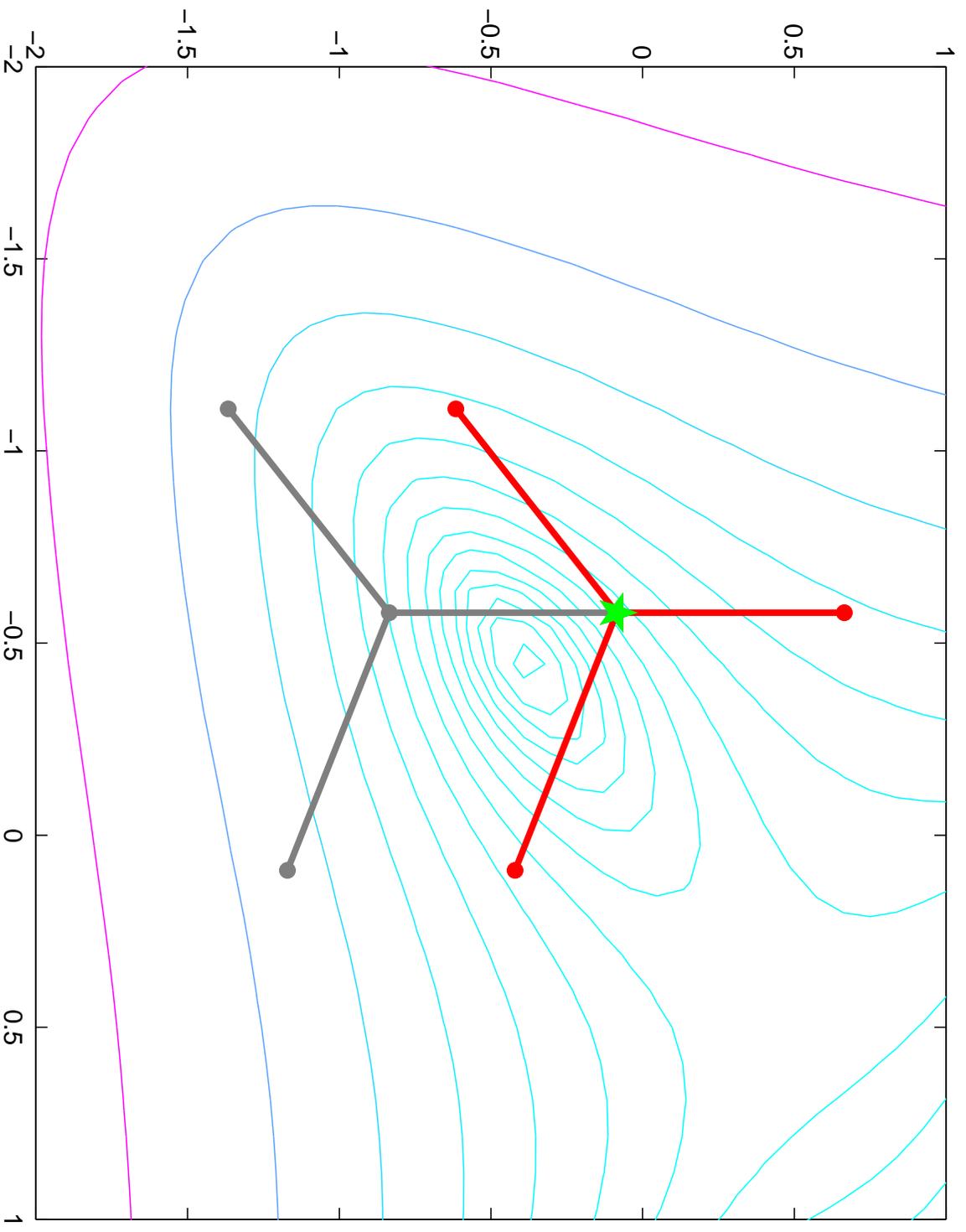
# PATTERN SEARCH



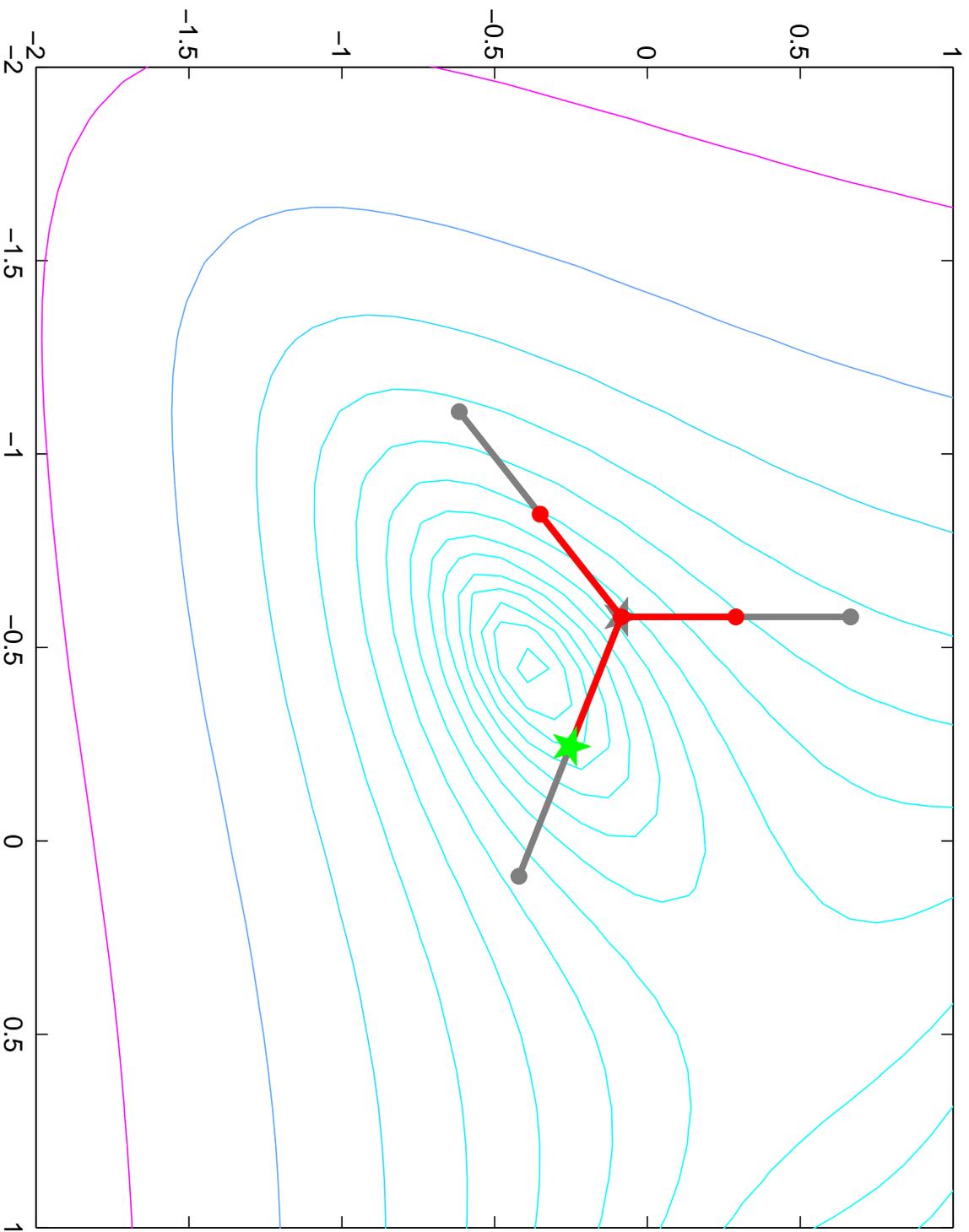
# PATTERN SEARCH



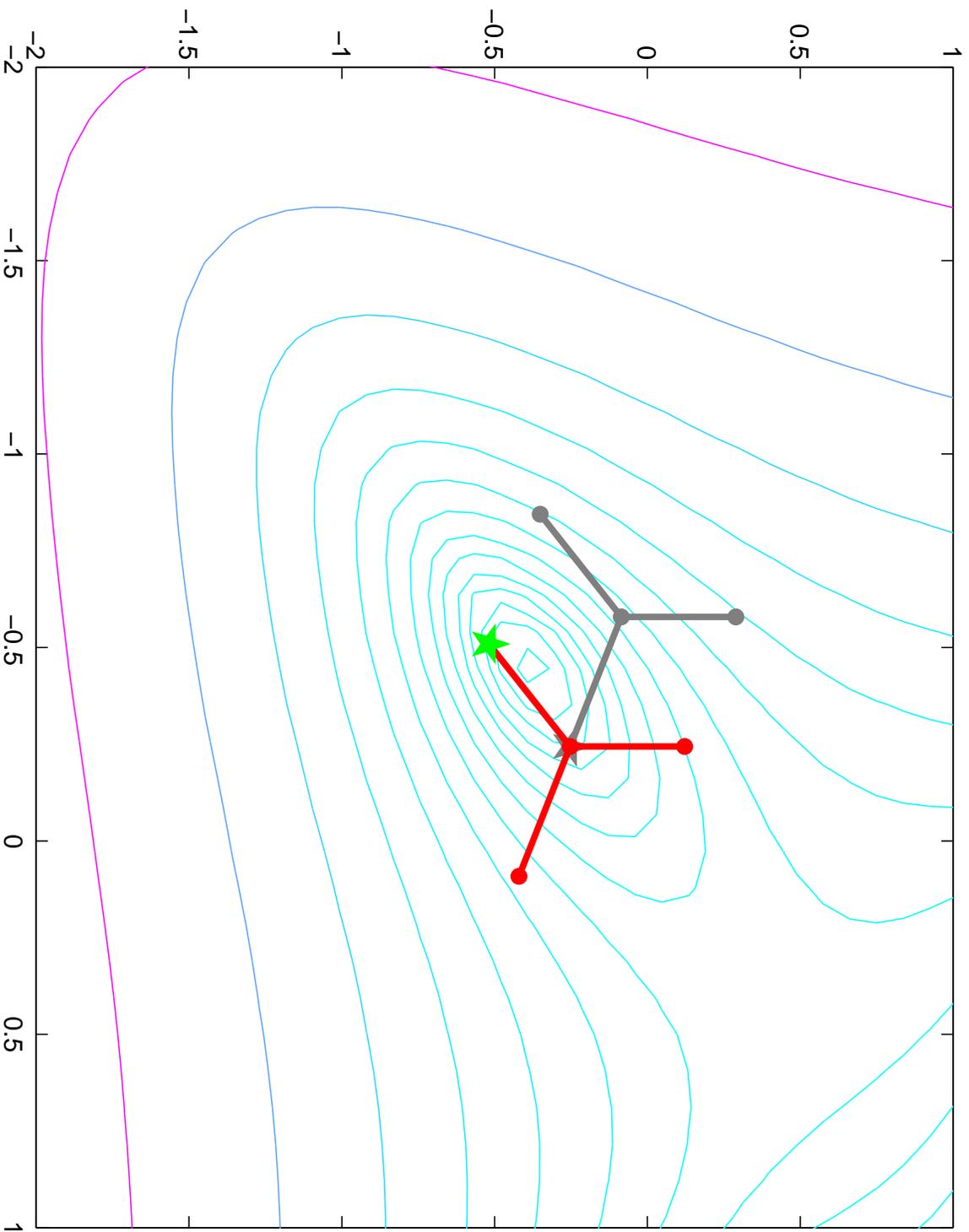
# PATTERN SEARCH



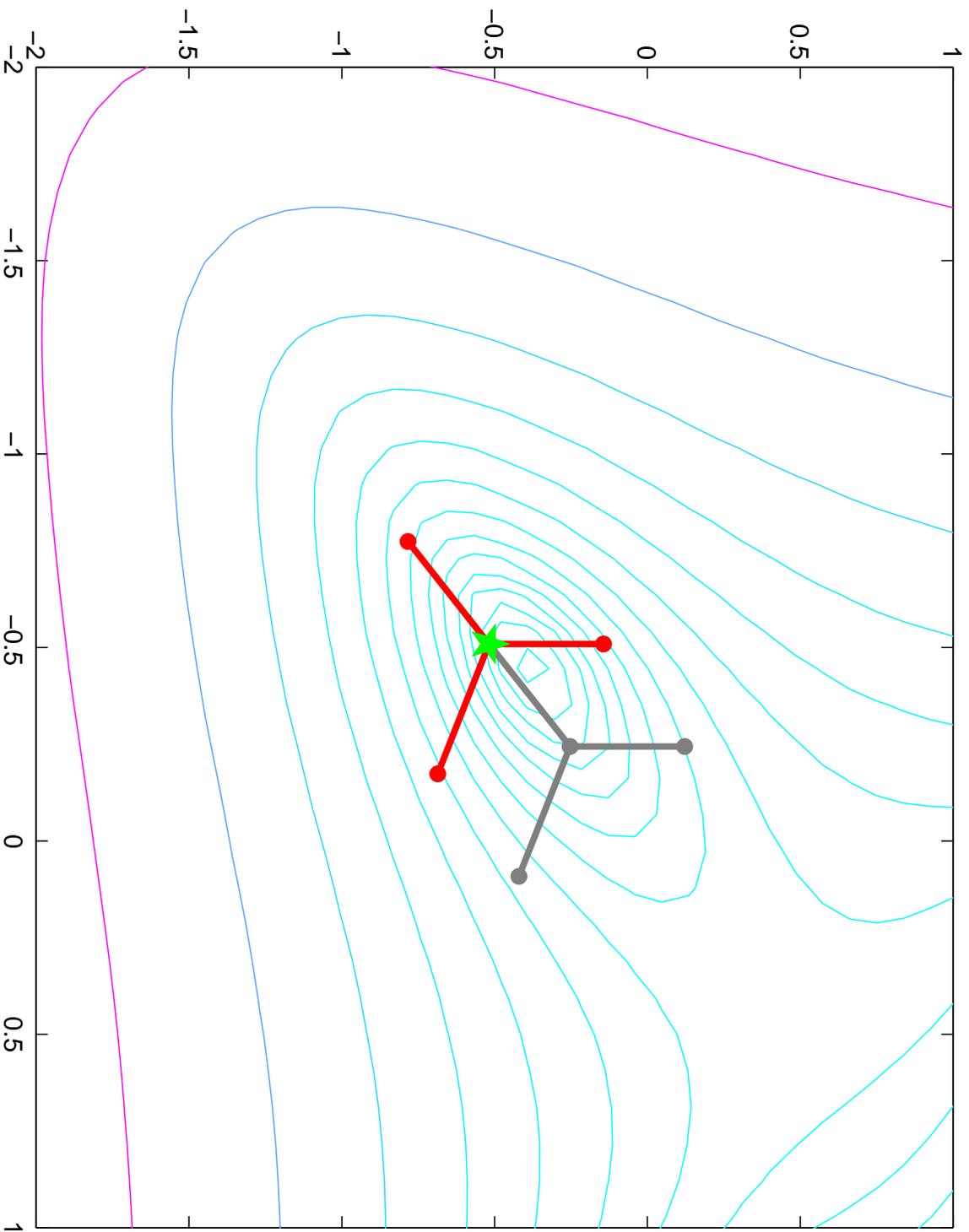
# PATTERN SEARCH



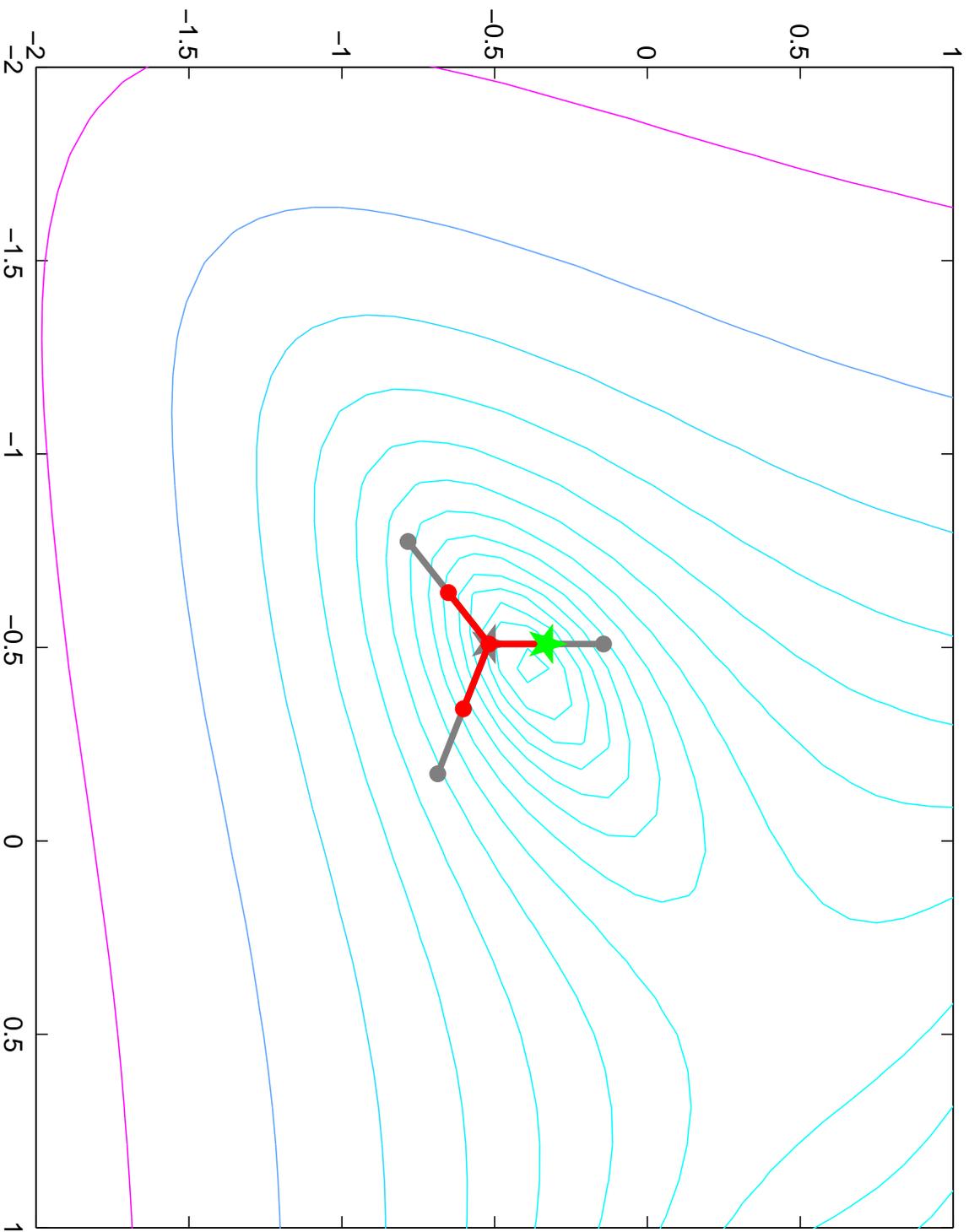
# PATTERN SEARCH



# PATTERN SEARCH



# PATTERN SEARCH



## CHOOSING THE SEARCH DIRECTIONS

The pattern must be chosen so that it positively spans  $\mathfrak{R}^n$ .

**Defn:** A set of vectors  $\{d_1, \dots, d_p\}$  *positively spans*  $\mathfrak{R}^n$  if any vector  $x \in \mathfrak{R}^n$  can be written as

$$x = \alpha_1 d_1 + \dots + \alpha_p d_p, \quad \alpha_i \geq 0 \quad \forall i.$$

That is, any vector can be written as a *nonnegative* linear combination of the basis vectors.

**Fact:** If  $\{d_1, \dots, d_p\}$  positively spans  $\mathfrak{R}^n$ , then for any  $x \neq 0$ , there exists  $d_i$  such that  $d_i^T x > 0$ .

# PARALLEL PATTERN SEARCH ALGORITHM

*This algorithm is from a single processor's point-of-view.*

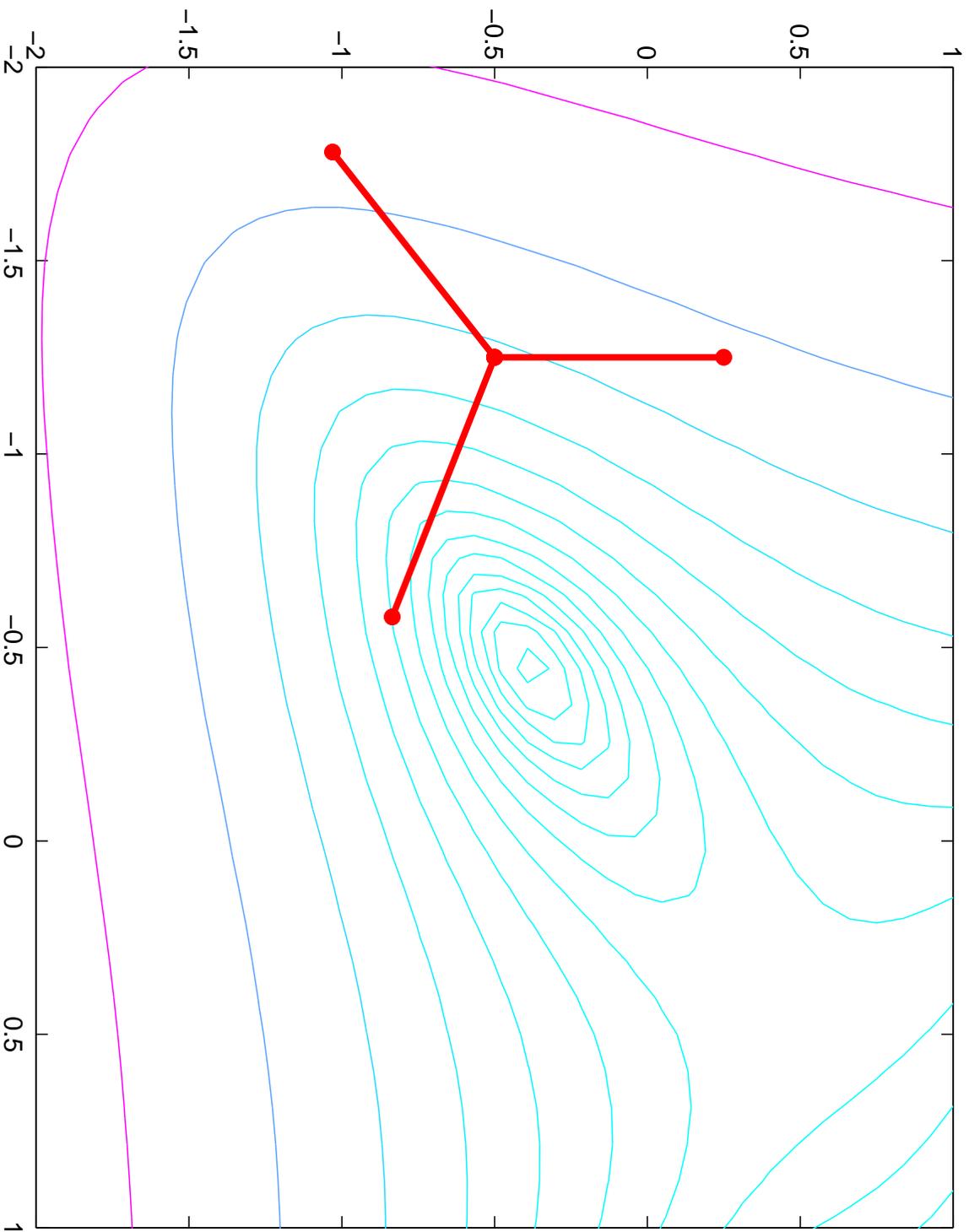
1. Compute  $x_{\text{trial}} \leftarrow x_{\text{best}} + \Delta_{\text{trial}} d$ , and evaluate  $f_{\text{trial}} \leftarrow f(x_{\text{trial}})$ .
2. Determine  $\{x_{\text{new}}, f_{\text{new}}\}$  via a **global reduction** on all  $\{x_{\text{trial}}, f_{\text{trial}}\}$  values.
3. If  $f_{\text{new}} < f_{\text{best}}$ , replace  $\{x_{\text{best}}, f_{\text{best}}\} \leftarrow \{x_{\text{new}}, f_{\text{new}}\}$ .  
Else  $\Delta_{\text{trial}} \leftarrow \frac{1}{2} \Delta_{\text{trial}}$ .
4. If  $\Delta_{\text{trial}} > \text{tol}$ , go to Step 1. Else, exit.

## PROBLEMS WITH PPS

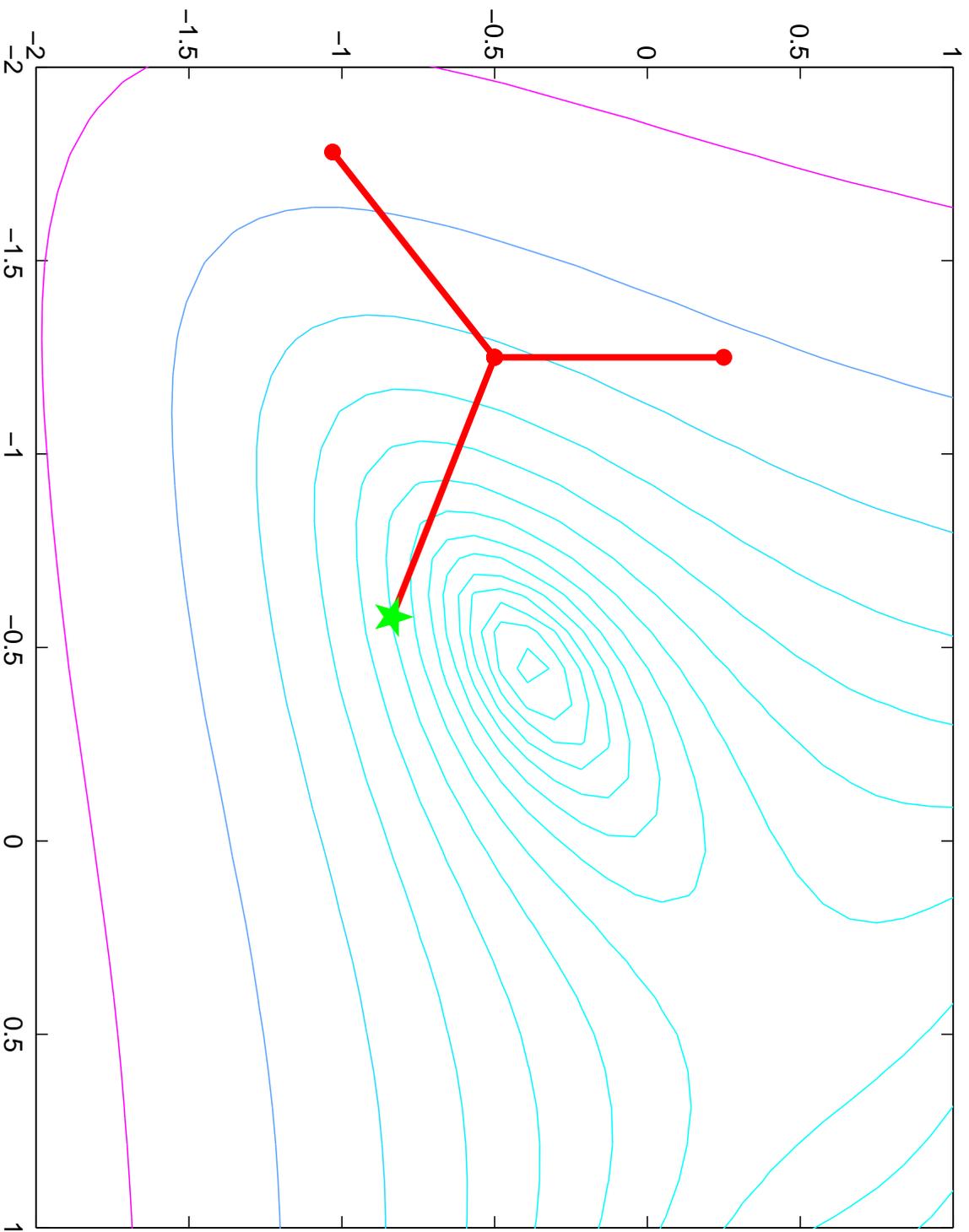
The global reduction in Step 2 is a potential *bottleneck*; that is, some processors may sit idle while waiting for others to finish. This can happen for several reasons...

1. The **computation time** for the objective function may vary depending on the inputs.
2. The **load** on the individual processes may vary.
3. Groups of processors participating in the calculation may possess varying **performance characteristics**.

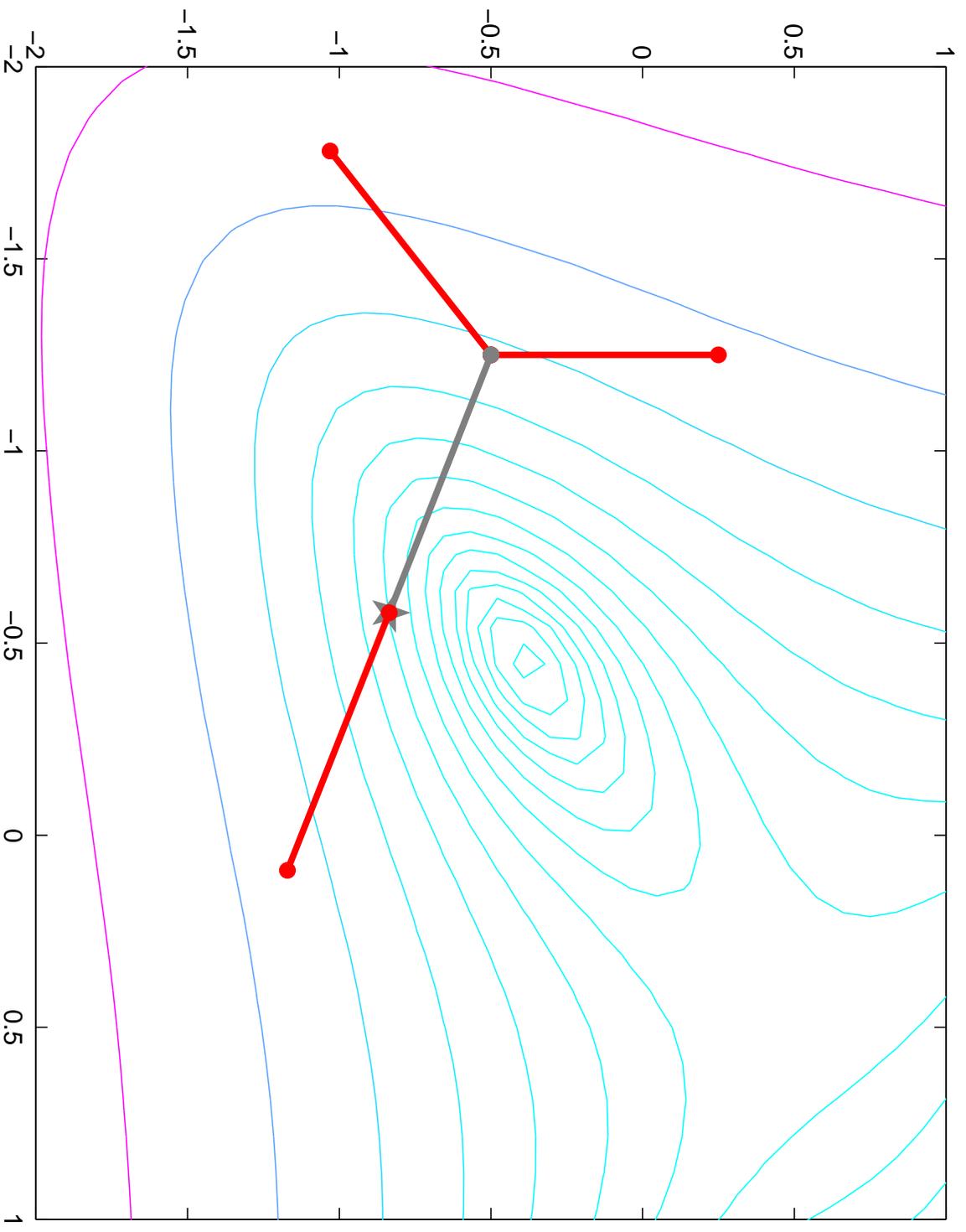
# ASYNCHRONOUS PARALLEL PATTERN SEARCH



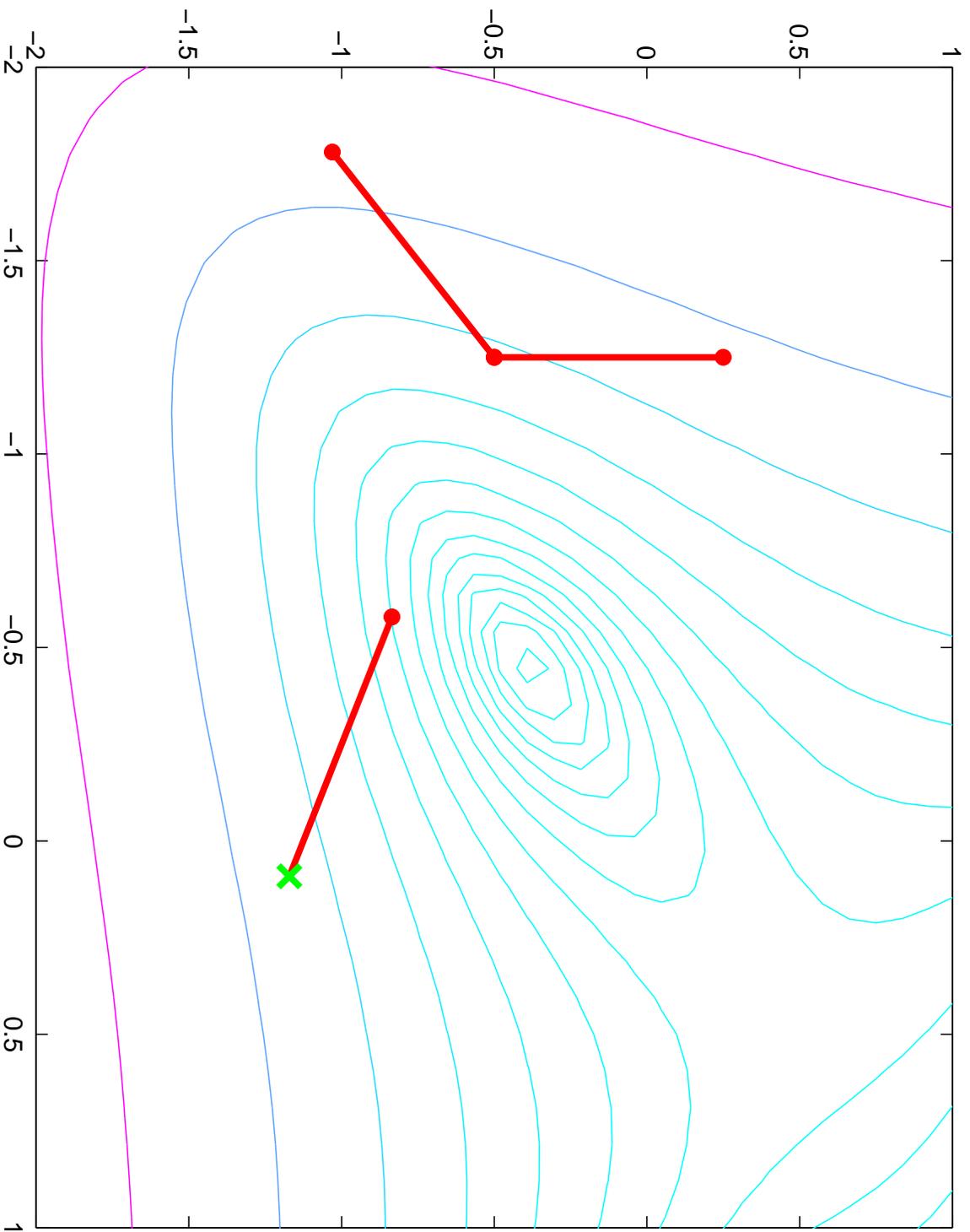
# ASYNCHRONOUS PARALLEL PATTERN SEARCH



# ASYNCHRONOUS PARALLEL PATTERN SEARCH

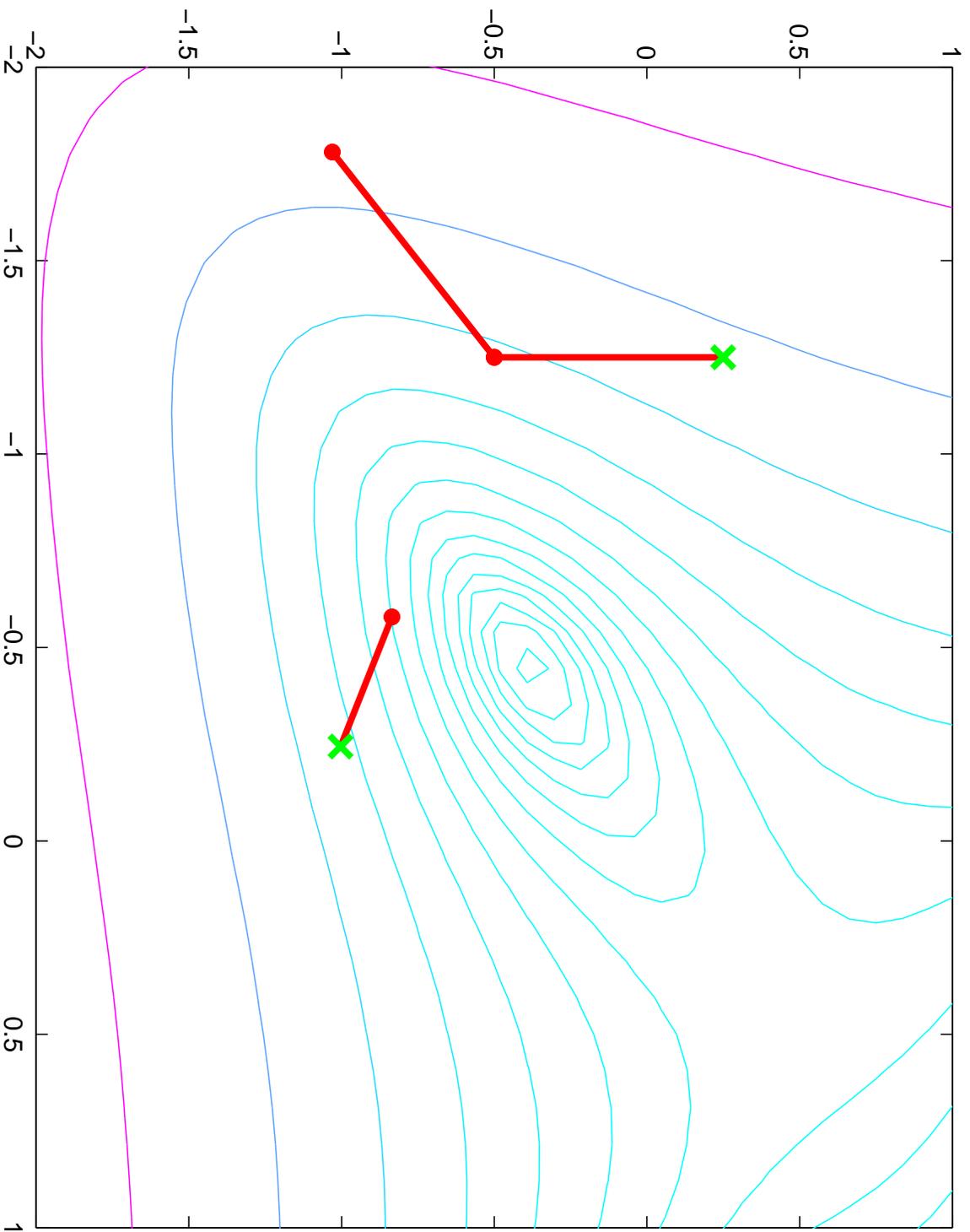


# ASYNCHRONOUS PARALLEL PATTERN SEARCH

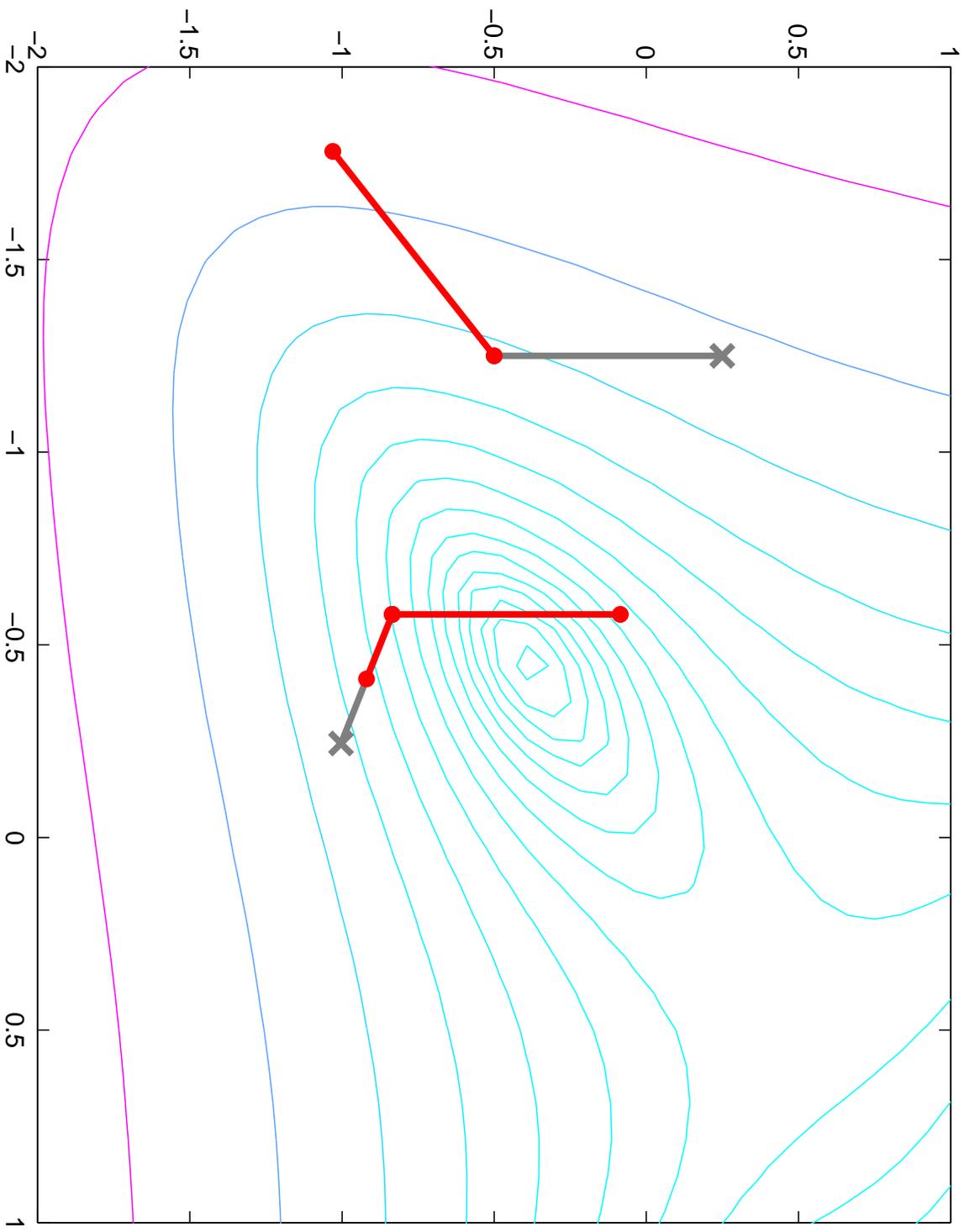




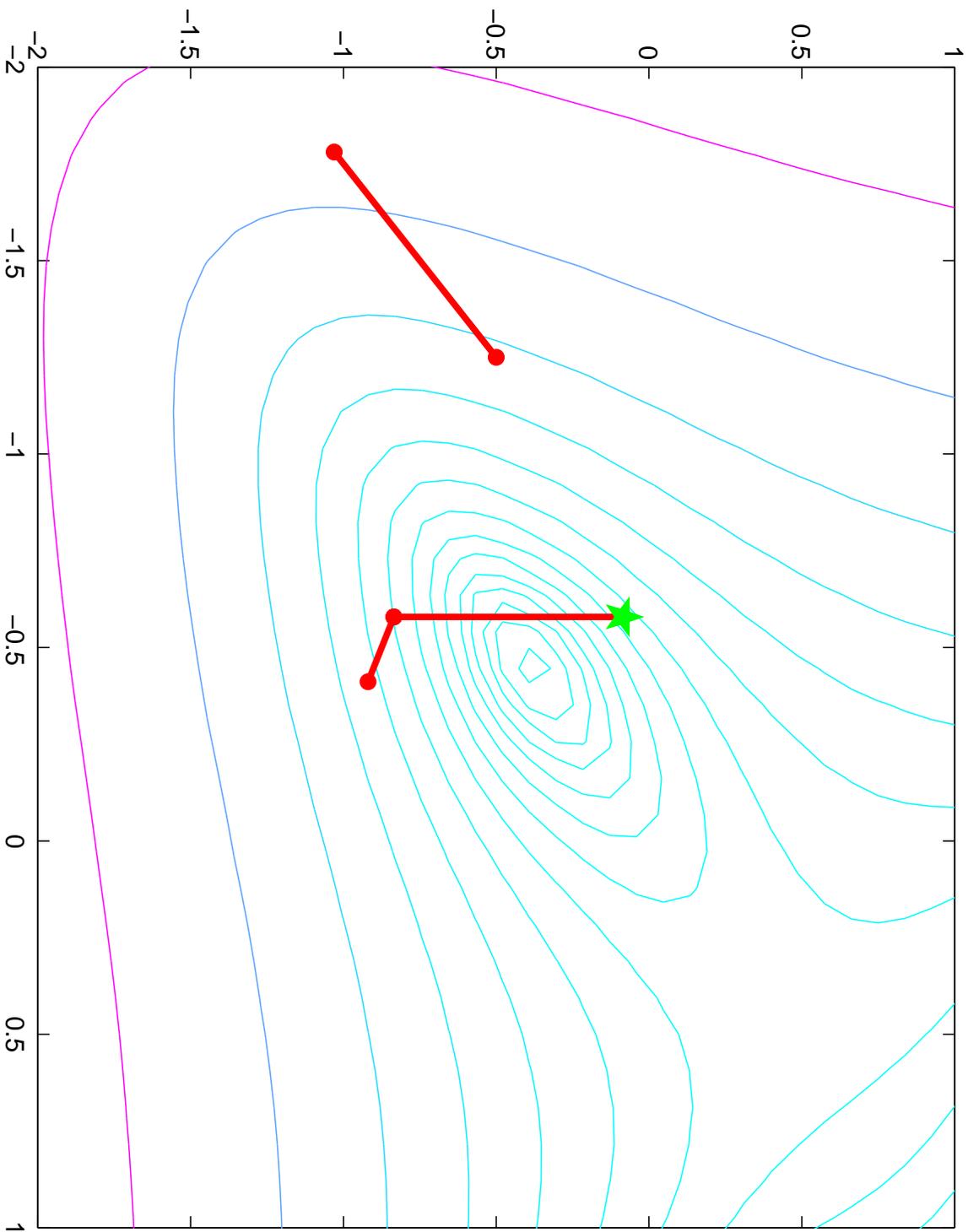
# ASYNCHRONOUS PARALLEL PATTERN SEARCH



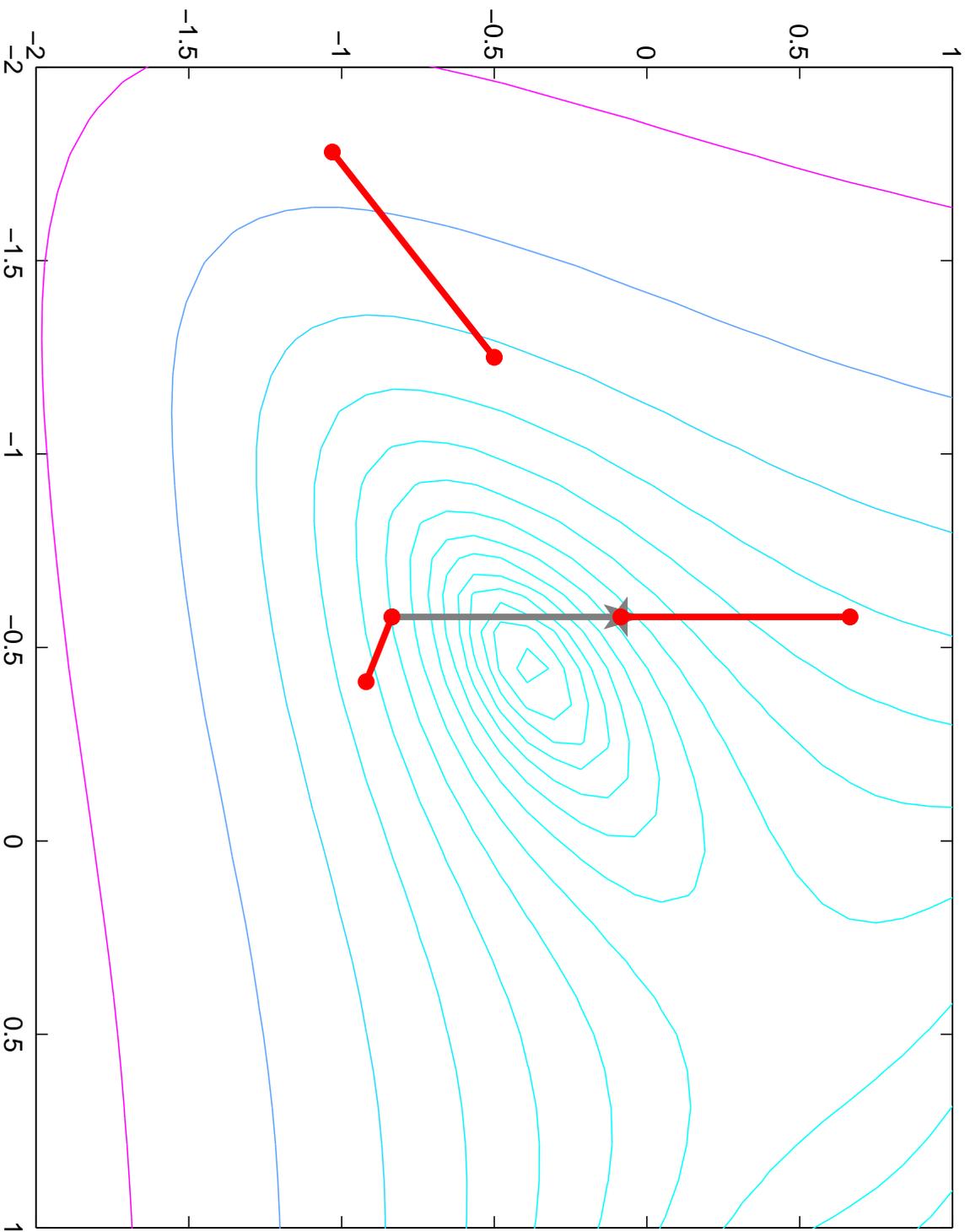
# ASYNCHRONOUS PARALLEL PATTERN SEARCH



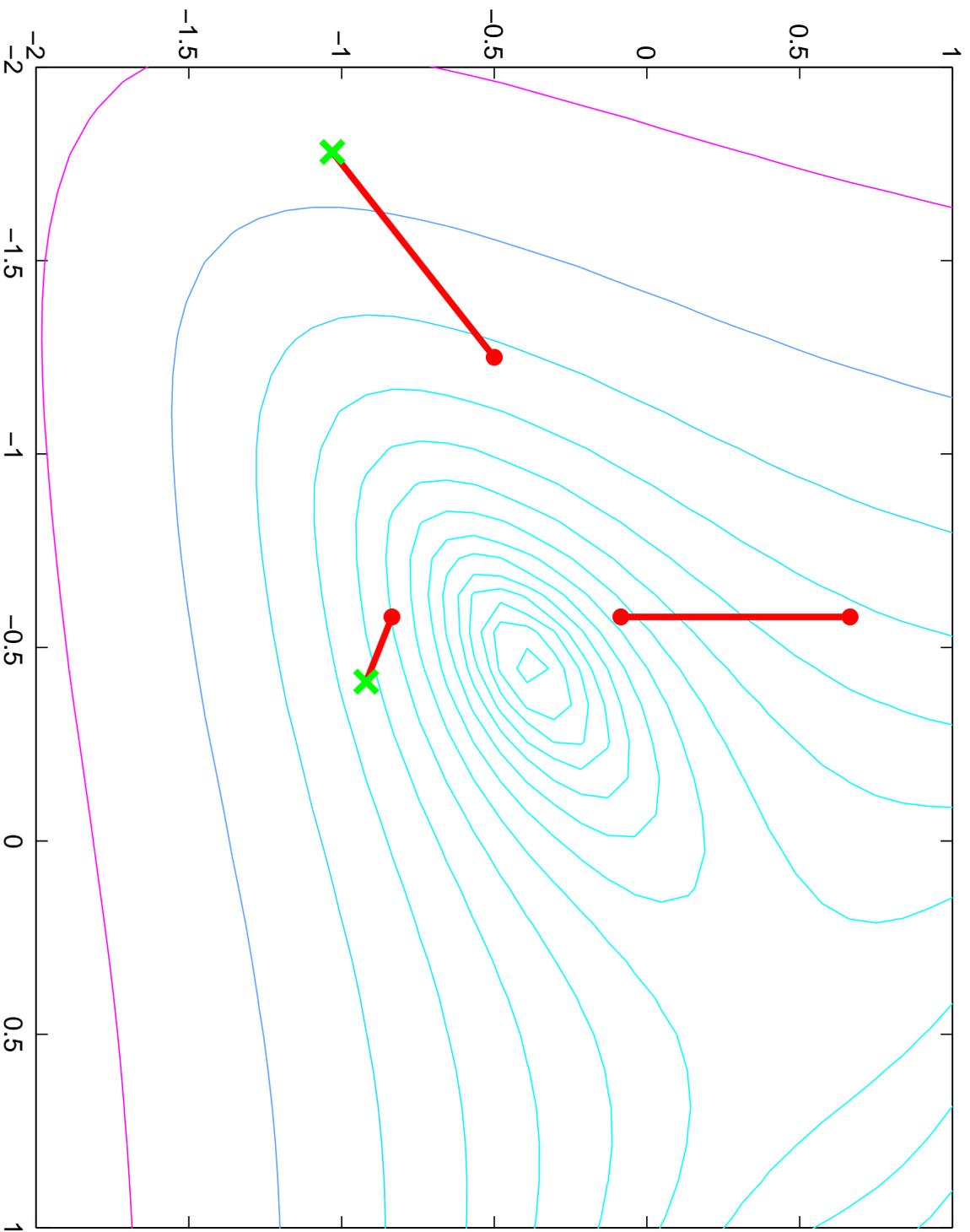
# ASYNCHRONOUS PARALLEL PATTERN SEARCH



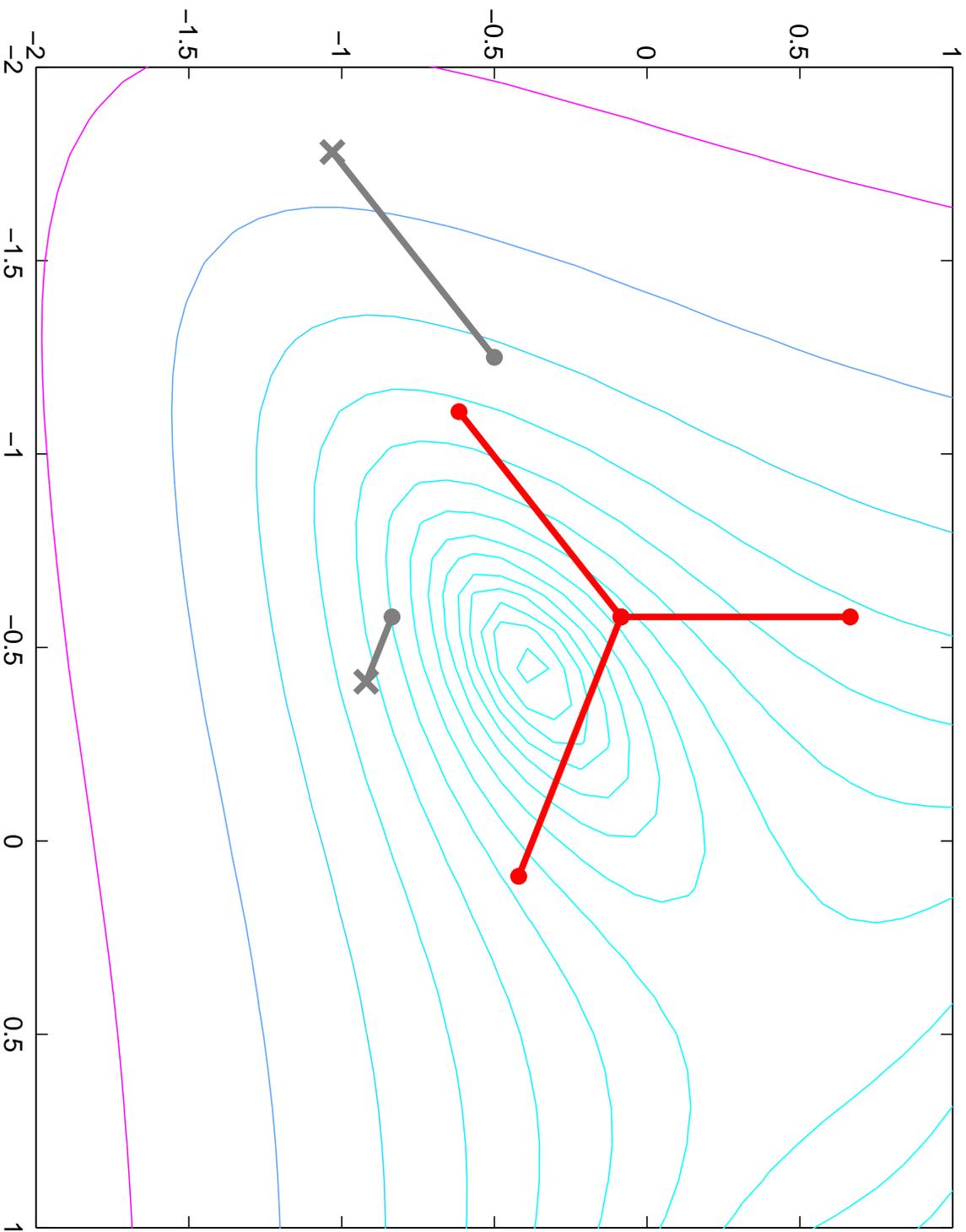
# ASYNCHRONOUS PARALLEL PATTERN SEARCH



# ASYNCHRONOUS PARALLEL PATTERN SEARCH



# ASYNCHRONOUS PARALLEL PATTERN SEARCH



# ASYNCHRONOUS PARALLEL PATTERN SEARCH

0. Consider each incoming triplet  $\{x_{\text{new}}, f_{\text{new}}, \Delta_{\text{new}}\}$  received from another processor. If  $f_{\text{new}} < f_{\text{best}}$ , replace  $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\} \leftarrow \{x_{\text{new}}, f_{\text{new}}, \Delta_{\text{new}}\}$ ,  $\Delta_{\text{trial}} \leftarrow \Delta_{\text{best}}$ .
1. Compute  $x_{\text{trial}} \leftarrow x_{\text{best}} + \Delta_{\text{trial}} d$ , and evaluate  $f_{\text{trial}} \leftarrow f(x_{\text{trial}})$ .
2. Set  $\{x_{\text{new}}, f_{\text{new}}, \Delta_{\text{new}}\} \leftarrow \{x_{\text{trial}}, f_{\text{trial}}, \Delta_{\text{trial}}\}$ .
3. If  $f_{\text{new}} < f_{\text{best}}$ , then  $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\} \leftarrow \{x_{\text{new}}, f_{\text{new}}, \Delta_{\text{new}}\}$ ,  $\Delta_{\text{trial}} \leftarrow \Delta_{\text{best}}$ , and *broadcast* the new minimum triplet  $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\}$  to all other processors. Else  $\Delta_{\text{trial}} \leftarrow \frac{1}{2} \Delta_{\text{trial}}$ .
4. If  $\Delta_{\text{trial}} > \text{tol}$ , goto Step 0. Else *broadcast* a local convergence message for the pair  $\{x_{\text{best}}, f_{\text{best}}\}$ .
5. Wait until either (a) enough of processes have converged for this point or (b) a better point is received. In case (a), exit. In case (b), goto Step 0.

# APPS DAEMON

- **New Minimum from Another Processor**  
If  $f_{\text{new}} < f_{\text{best}}$ , then  $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\} \leftarrow \{x_{\text{new}}, f_{\text{new}}, \Delta_{\text{new}}\}$ .
- **Return from function evaluation**  
If  $f_{\text{trial}} < f_{\text{best}}$ , then  $\{x_{\text{best}}, f_{\text{best}}, \Delta_{\text{best}}\} \leftarrow \{x_{\text{trial}}, f_{\text{trial}}, \Delta_{\text{trial}}\}$ ,  
 $\Delta_{\text{trial}} \leftarrow \Delta_{\text{best}}$ , **broadcast the new minimum, and spawn a new function evaluation.**  
Else if  $x_{\text{best}} \neq \text{trial point generator}$ , then spawn a new function evaluation using  $\Delta_{\text{trial}} \leftarrow \Delta_{\text{best}}$ .  
Else  $\Delta_{\text{trial}} = \frac{1}{2} \Delta_{\text{trial}}$ . If  $\Delta_{\text{trial}} < \text{tol}$ , then broadcast a local convergence message, else spawn a new function evaluation.
- **Local Convergence Message**  
Go through steps for new minimum to verify point. If incoming point is the same as best, then update its *convergence table* and check for *convergence*.

PLAY APPS MOVIE

## ARCHITECTURE: PVM

PVM stands for *Parallel Virtual Machine* and is a communication architecture for parallel and distributed computing.

PVM is particularly useful in cluster computing because it can be used in a heterogeneous environment and supports fault tolerance.

PVM provide the following capabilities:

- Blocking and non-blocking sends and receives (including message tags and probe ability)
- Process spawning and termination ability
- Ability to detect task failures
- Access to host configuration information

PVM's weakness is its dependence on the master daemon to survive.

## EXAMPLE: A THERMAL DESIGN PROBLEM

- **Objective:** Determine *power settings* for heaters on a thermal deposition furnace for silicon wafers
- **Simulation Code:** TWAFER yields measurements at a discrete collection of points on the wafers

$$f(x) = \sum_{j=1}^N (T_j(x) - T_*)^2$$

$x$  = Unknown power settings

$T_j(x)$  = Temperature at measurement point  $j$

$N$  = Total number of measurement points

$T_*$  = Ideal temperature

## NUMERICAL RESULTS – FOUR HEATERS

- Search directions - Regular Simplex (5) plus Random (15)
- $\Delta = 10.0$ , and  $tol = 0.1$ .
- Each function evaluation takes approximately 1.3 seconds
- Quick fix for bound constraints—interior solution
- Average of 10 runs

Method	Procs	$f(\mathbf{x}^*)$	Function Evals	Idle Time	Total Time
APPS	20	0.67	334.6	0.17	395.94
PPS	20	0.66	379.9	44.77	503.88

## NUMERICAL RESULTS – SEVEN HEATERS

- Search directions - Regular Simplex (8) plus Random (27)
- $\Delta = 10.0$ , and  $tol = 0.1$ .
- Each function evaluation takes approximately 10.4 seconds
- Quick fix for bound constraints—interior solution
- Average of 9 runs

Method	Procs	f(x*)	Function Evals	Idle Time	Total Time
APPS	35	3.30	240.4	71.48	2260.46
PPS	35	2.85	202.2	213.90	2306.83

# EXAMPLE: AN ELECTRICAL CIRCUIT SIMULATION

- **Variables:** inductances, capacitances, diode saturation currents, transistor gains, leakage inductances, and transformer core parameters
- **Simulation Code:** SPICE3

$$f(x) = \sum_{t=1}^N (V_t^{\text{SIM}}(x) - V_t^{\text{EXP}})^2,$$

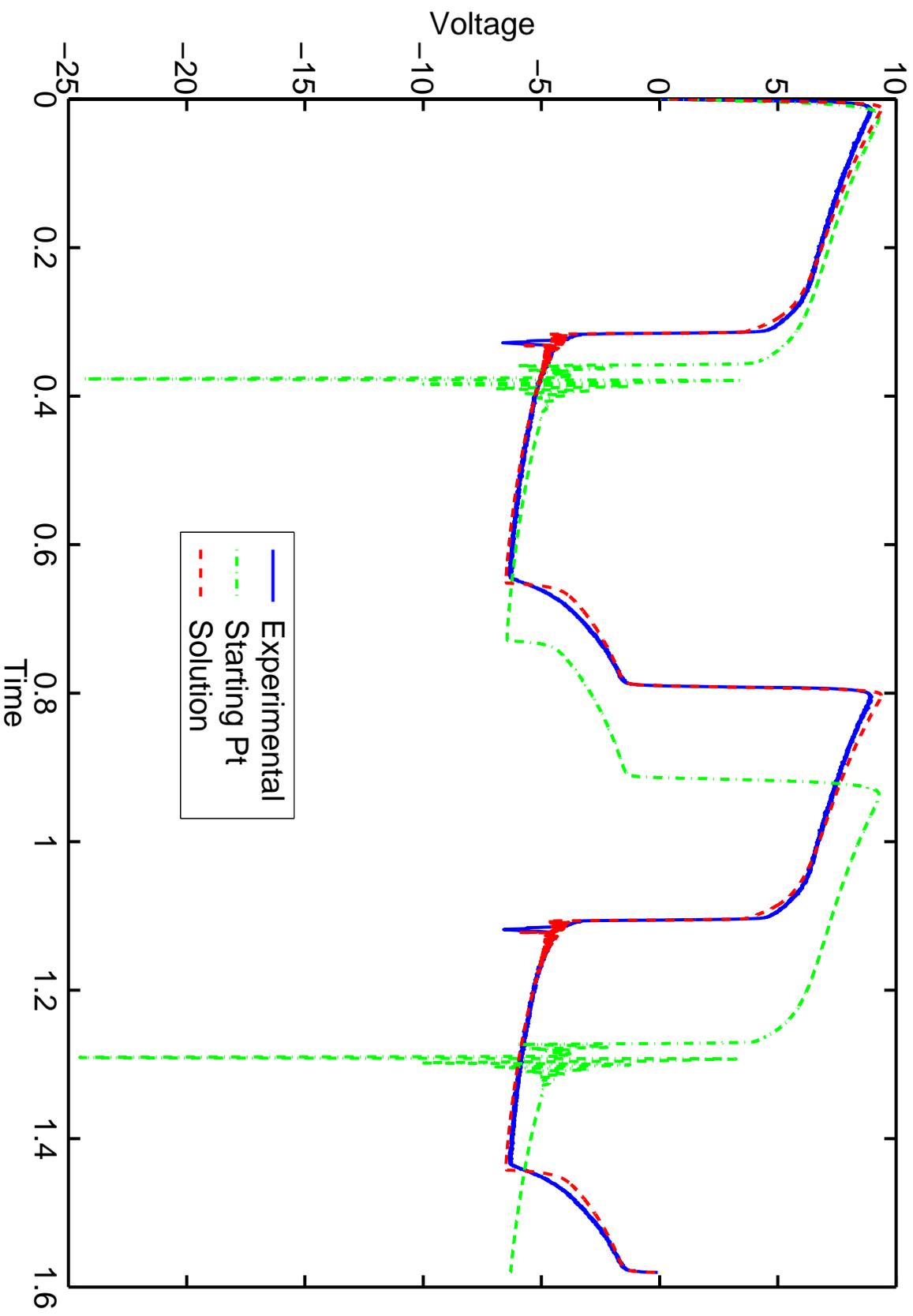
$x$  = 17 unknown characteristics

$V_t^{\text{SIM}}(x)$  = Simulation voltage at time  $t$

$V_t^{\text{EXP}}$  = Experimental voltage at time  $t$

$N$  = Number of timesteps

# CIRCUIT PROBLEM RESULTS



# CIRCUIT PROBLEM NUMERICAL RESULTS

- Search directions are  $\pm$  Unit Vectors (34) plus Random
- $\Delta = 1.0$ , and  $tol = 0.1$ .
- Each function evaluation takes approximately 20 seconds
- Quick fix for bound constraints—solution on boundary

Method	Procs	$f(\mathbf{x}^*)$	Function Evals	Idle Time	Total Time
APPS	34	26.2	57.5	111.92	1330.55
APPS	50	26.9	50.6	63.22	807.29
PPS	34	28.8	53.0	521.48	1712.24
PPS	50	34.9	47.0	905.48	1646.53

## FAULT TOLERANCE

- We can afford to lose processes as long as we maintain a positive basis.
- Processes can be restarted when we no longer have a positive basis.
- Progression towards a solution continues as long as one or more APPS processes is alive.
- Completion does *not* depend on the survival of any particular APPS daemon.

# PLAY APPS WITH FAULTS MOVIE

## EXAMPLE II REVISITED

The “fault” versions have a failure in the function evaluation or daemon every 30 seconds.

<b>Initial Procs</b>	<b>Final Procs</b>	<b>f(x*)</b>	<b>Total Time</b>
34	34	26.2	1330.55
34-faults	34	27.8	1618.46
50	50	26.9	807.29
50-faults	32	54.2	1041.14

*In the 50-faults results, the “quick fix” for bound constraints comes back to haunt us.*

# CONVERGENCE THEORY FOR APPS

Assume that  $f$  is continuously differentiable and that the level set  $\mathcal{L}(x_0)$  is bounded. Further assume that the maximum message time is bounded above and that the maximum function evaluation time is bounded above.

**Theorem.** There exists a subsequence  $T'_i$  such that

$$\lim_{\substack{t \rightarrow +\infty \\ t \in T'_i}} \Delta_i^t = 0, \quad i = 1, \dots, p$$

**Theorem.** There exists a subsequence  $T''_i \subseteq T'_i$  such that

$$\lim_{\substack{t \rightarrow +\infty \\ t \in T''_i}} x_i^t = \hat{x}_i, \quad i = 1, \dots, p$$

Furthermore,  $\hat{x}_i = \hat{x}_j$  for all  $i \neq j$ .

**Theorem.** At the limit point  $\hat{x}$  ( $= \hat{x}_i$ ),

$$\nabla f(\hat{x}) = 0$$

## CONCLUSIONS

- Introduced APPS, an asynchronous and fault tolerant parallel pattern search method for optimization.
- APPS is useful when function evaluation time varies or faults are a concern.
- APPS has proportionally much less idle time than PPS, and APPS was demonstrated to be faster than PPS on two engineering problems.
- APPS incorporates a very high level of fault tolerance at both the algorithmic and the coding level.

## FUTURE WORK

- Constraints (bounds, linear, nonlinear)
- Function value cache
- Conditioning of positive basis
- Dynamic search direction based on model
- Pure MPI Version (No Fault Tolerance)

## HIGHLIGHTS OF LATEST INCARNATION

- One APPS Agent Per Host (less overhead)
- May Specify “Ideal” Number of Reveals for Each Host
- May Assign 0 Function Evaluations to a Host
- Dynamic Addition/Deletion of Hosts
- “Smart” Remapping of Reveals to Hosts (with several options)
- Better Handling of Failed Reveals

## LINKS

<http://csmr.ca.sandia.gov/projects/apps.html>

<http://csmr.ca.sandia.gov/~tgkolda/pubs.html#apps>