Operating System Support for Resilience*

TR11-003

Kathleen McGill, *Student Member, IEEE,* and Stephen Taylor, *Member, IEEE*

Thayer School of Engineering, Dartmouth College, 8000 Cummings Hall, Hanover, NH 03755

Kathleen.N.McGill@dartmouth.edu, Stephen.Taylor@dartmouth.edu

Keywords: Application resilience, software reliability, load balancing, robotic swarming algorithms, heat diffusion algorithms

Abstract

This paper is concerned with improving the resilience of mission-critical applications to a wide variety of failures, errors, and malicious attacks. A number of approaches have been proposed in the literature based on fault tolerance provided through replication of resources. In general, these approaches provide graceful degradation of performance to the point of failure but do not *guarantee* progress in the presence of multiple cascading and recurrent attacks. Our approach is to dynamically replicate processes, detect inconsistencies in their behavior, and transparently restore the level of fault tolerance as a computation proceeds. This approach is achieved through a collection of *operating system mechanisms* for process replication, migration,

*This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-09-1-0213. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

and communication.

This paper describes a novel concurrent scheduling algorithm that manages replicated processes, inspired by the notions of *heat diffusion* and *robotic swarming*. Heat diffusion is emulated to disseminate processes across computer architecture. Robotic swarming techniques are used to *maintain locality* between replicated processes while balancing load. To quantify the performance of the algorithm, we compare it to three algorithms from the robotics literature. The basis for comparison is a large scale benchmark set that we have devised, with an associated sensitivity analysis. The benchmarks were designed to fill a gap in the literature allowing direct, repeatable comparisons between competing algorithms.

I. Introduction

Commercial-off-the-shelf (COTS) computer systems have traditionally provided several measures to protect organizations from hardware failures, such as RAID file systems [1] and redundant power supplies [2]. Unfortunately, there has been relatively little effort to provide similar levels of fault tolerance to software errors and exceptions. In recent years, computer network attacks have added a new dimension that decreases overall system reliability. A broad variety of technologies have been explored for detecting these attacks using intrusion detection systems [3]-[5], file-system integrity checkers [6]-[7], rootkit detectors [8]-[11], and a host of other technologies. Unfortunately, creative attackers and trusted insiders have continued to undermine confidence in software. These robustness issues are magnified in distributed applications, which provide multiple points of failure and attack.

Our approach is to dynamically replicate processes, detect inconsistencies in their behavior, and transparently restore the level of fault tolerance as the computation proceeds [12]. Figure 1 illustrates how this strategy is achieved. At the application level, three communicating processes

share information using message-passing. The underlying operating system implements a resilient view that replicates each process and organizes communication between the resulting *process groups*. Individual processes within each group are mapped to different computers to ensure that a single attack or failure cannot impact an entire group. The base of the figure shows how the process structure responds to attack or failure. The figure assumes that an attack is perpetrated against processor 3, causing processes 1 and 2 to fail or to portray communication inconsistencies with other replicas within their group. Failures are detected by timeouts and/or message comparison. These failures trigger automatic process regeneration; the remaining consistent copies of processes 1 and 2 dynamically regenerate a new replica and migrate it to processors 4 and 1 respectively. As a result, process resiliency is reconstituted, and the application continues operation with the same level of assurance.

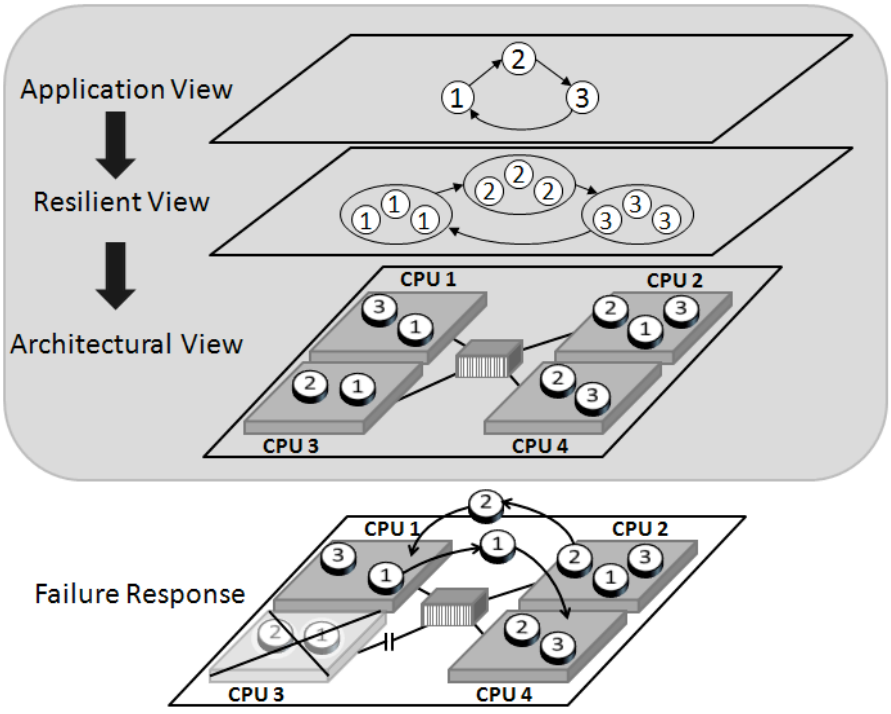


Fig. 1. Dynamic process regeneration.

This approach requires several new operating system mechanisms. Process *replication* is needed to transform single processes into process groups. Process *mobility* is required to move a process from one processor to another. As processes move around the architecture, it is necessary to provide control over where processes are *mapped*. Point to point communication between application processes must be replaced by group communication between process groups. In order to prevent prohibitive communication costs, it is desirable to *maintain locality* within process groups. In the presence of locality, we expect replicated messages within a process group to be received at approximately the same time. Therefore, message transit delays from earlier messages can be used to set an upper bound on the delay for failure detection timeouts. Finally, mechanisms to detect process failures and inconsistencies must be available to initiate process regeneration.

This paper describes a distributed process scheduling algorithm, DIFFUSE, that disperses processes for the purpose of load balancing but maintains locality between replicas to bound transit delays. The algorithm is inspired by the notions of *heat diffusion* and *robotic swarming*: Heat diffusion is emulated to distribute processes associated with multiple computations across computer architecture [13]-[14]. Robotic swarming techniques are used to maintain locality between replicated processes. To quantify the performance of the algorithm, we compare it with three algorithms from the robotics literature. These algorithms, termed Biased Random Walk (BRW) [15], Glowworm Swarm Optimization (GSO) [16]-[19], and a GSO/BRW hybrid [20], are particularly attractive due to their simplicity, communication locality, and scalability. The basis for our comparison is a large scale benchmark set that we have devised with an associated sensitivity analysis, based on previous studies [20]-[21]. The benchmarks were designed to fill a gap in the literature allowing direct, repeatable comparisons between competing algorithms [22].

Our DIFFUSE algorithm outperforms the robotics algorithms and may serve as the basis for distributed operating system support for resilience.

II. Related Research

Distributed process scheduling is a well-established field. Diffusive algorithms [13], [14], [23]-[28], dimension exchange methods [29]-[31], and gradient models [32]-[33] are among the most prevalent approaches. Heirich and Taylor proposed a parabolic diffusion scheme based on the heat diffusion equation [14]. This model characterizes process load as a scalar *heat* value and automatically disperses processes in the architecture. This approach has several attractive properties: It is a simple, scalable algorithm that uses a completely local iteration and only nearest neighbor communication. It provides global convergence to a balanced CPU load and provides guarantees for global convergence and progress through well-established mathematical analysis. The algorithm has been shown, through simulation, to simultaneously balance multiple independent load distributions over large-scale architectures representing concurrent computations injected at random locations with disparate random loads [14]. Extensions to the algorithm allow multiple properties, including communication, memory, and CPU load, to be balanced simultaneously [28].

Robotic swarming algorithms present an alternative view of resource management in which swarms of distributed and autonomous agents achieve a common goal while imposing emergent properties of *swarm cohesion* and *obstacle avoidance*. There is a large field of research that utilizes robotic swarms for emission source localization; in a separate publication, we have presented a survey of this research [22]. In emission source localization, robots sample the *gradient* of some physical or chemical property in order to locate potential sources. A large number of robots, equipped with sensors and inter-robot communication, may collectively search

for all sources in minimal time. This problem is analogous to the load balancing problem in that a large number of processes sample load and search for troughs in processor utilization. As a result, we perceive a direct application of robotic swarming algorithms in distributed process scheduling.

There are several classes of algorithms in the robotics literature that are candidates for process scheduling: 1) biologically-inspired approaches based on *E. Coli* bacteria [15], glowworms [16]-[19], and other social foraging organisms [34], [35]; 2) population- and evolutionary-based models, including genetic algorithms [36] and Particle Swarm Optimization [37], [38]; and 3) probabilistic models based on Bayesian occupancy grid mapping [39]. Two approaches from this body of work stand out as potential candidates for load balancing: the BRW algorithm [15] and the GSO algorithm [16]-[19].

The BRW algorithm [15], shown in Figure 2, is inspired by the chemotaxis of *E. Coli* bacteria and consists of two actions: a *run* and a *tumble.* A run represents a process movement in a straight line, and a tumble is a random reorientation in a new direction. The presence of a load gradient affects the length of the BRW process's run. If a run is in the direction of a positive gradient, the length of the run is extended by a bias step. By extending the length of runs in the direction of the positive gradient, the process gradually moves toward the gradient source. The BRW pseudocode in Figure 2 has a step length of ten units and a 10% bias. Processes conducting a BRW have no prior knowledge of the architecture. The BRW stands out in the literature as particularly valuable for comparative analysis. It provides autonomous process mobility *without communication*. In addition, BRW is simple to implement and has successfully located multiple disparate gradient sources in a variety of simulations [15].

```
bias = 10%;
step_length = 10;
step_extension = step_length*bias;
C1 = measure_local_resources();
while (target_not_found)  {
     direction = tumble();
     run(direction, step_length);
     C2= measure_local_resources();
     if( C2 > C1)  {
          run(direction, step_extension);
          C2= measure_local_resources();
     }
     C1 = C2;
}
```

Fig. 2. BRW pseudocode for run and tumble sequence with 10% bias [15].

The GSO algorithm [16]-[19] models processes as glowworms that possess a luminescence quantity called *luciferin* that causes them to glow in response to the local field, corresponding to the level of CPU resources available. In nature, female worms glow to attract mates, but, in the GSO algorithm, the glow attracts other processes to engage in a cooperative search. Each process has a *limited communication range* and an adaptive decision range that is less than or equal the process's communication range. A process selects neighbors that are within its decision range and have higher luciferin values. To begin a search for resources, a process chooses a leader from its neighbors and moves toward it. The most probable choice for the leader is the neighbor

with the highest luciferin value, corresponding to the likely direction of a load trough, or a cluster of under-utilized computers. As a result of this leader selection, subgroups form within the swarm and begin searching for nearby load troughs. The GSO's adaptive decision range is a vital feature of the algorithm. It provides the ability to partition the swarm, without global communication, to search for multiple load troughs simultaneously. This capability is well suited for the load balancing problem in which the goal is to utilize the entire system.

In the past, the robotics literature lacked a common set of validation benchmarks, making it impossible to directly compare algorithms and weigh their merits for different applications. To solve this problem, we introduced a set of benchmark cases to provide *ground-truth* and conducted a comparative analysis of the BRW and GSO algorithms using these benchmarks [20].

This research uncovered a potential weakness in the GSO algorithm. If a GSO process has no neighbors within communication range to select as a leader, it will not move. As a result, processes with an empty neighborhood will not contribute to the search for resources. We explored a GSO/BRW hybrid algorithm to address this limitation. The hybrid process performs the same actions as the original GSO algorithm, unless the process's neighborhood is empty. In this case, the process performs a single BRW step.

After exhaustive analysis, we concluded that none of these algorithms were successful in locating all load troughs in our benchmarks [20]. We identified several desired features to improve in performance: First, the algorithm should have a mechanism to search for resources when there is no load gradient to exploit. Second, processes should balance local search for resources with broad exploration of the architecture. Third, the processes should be confined within the dimensions of the architecture, taking appropriate actions at physical boundaries. Finally, processes should continue to aid other processes after locating a load trough [20].

III. DIFFUSE Algorithm

Figure 3 presents a new swarming algorithm, DIFFUSE, for distributed process scheduling. The algorithm includes two distinct modes of swarm control: SEARCH mode and DISPERSE mode. In SEARCH mode, processes conduct an independent *local* search for resources using the BRW algorithm with a 10% bias [15]. In DISPERSE mode, the processes react to virtual repulsive forces from neighbors and spread across the architecture [40]. Each process begins in DISPERSE mode to promote initial coverage of the architecture (1). Each iteration, the process checks for a load trough in its location (2) and shares its position and mode with all neighbors within communication range R (3). The process updates its mode based on the most recent communications (4) and implements either the SEARCH (5) or DISPERSE (6) algorithm. As a result, the processes' local interactions and random motions emerge as a collective *diffusion* of the swarm across the architecture.

```
mode = DISPERSE;                    /*1*/
while(troughs_not_found)  {
    check_for_troughs();            /*2 */
    neighbor_communication();       /*3*/
    update_mode();                  /*4*/
    if(mode = = SEARCH)  {          /*5*/
        BRW();
    }
    if(mode = = DISPERSE)  {        /*6*/
        calculate_net_force(net_force);
        move(net_force);
    }
}
```

Fig. 3. DIFFUSE algorithm pseudocode.

Specific conditions prompt a process to change its mode. If the process finds a cluster of under-utilized processor, it will remain in place and signal DISPERSE mode to its neighbors. If the process is in DISPERSE mode and in the same position as the last iteration, it changes to SEARCH mode. If the process is in SEARCH mode and communicates with a neighbor in DISPSERSE mode, the process changes to DISPERSE mode. Finally, if the process attempts to leave the architecture boundary at any time, it is reflected back into the domain.

These conditions combine the SEARCH and DISPERSE modes for effective localization of under-utilized processors. The DISPERSE mode signal originates from a process that first locates a load trough, and the signal propagates throughout the swarm. The switch to SEARCH

mode occurs when a process experiences no net force, either because it is equally spaced between neighbors or because there are no other processes within range. In the first case, the SEARCH initiates a local search for resources. In the second case, the SEARCH enables independent exploration. Upon switching to SEARCH mode, the process stays in SEARCH mode for a minimum number of BRW steps to permit sufficient local search for resources.

The DISPERSE mode algorithm creates a virtual electrostatic potential field in the architecture [40]. Unlike the BRW, GSO, and GSO/BRW hybrid algorithms, the DIFFUSE algorithm includes *a priori* knowledge of the architecture boundaries in order to confine processes. The processes and the boundary of the system are modeled as positively charged particles that repulse each other. The net force felt by an individual process, $\mathbf{F}_i$, is a summation of the individual repulsive forces from all neighbors, *j*, within communication range.

$$\mathbf{F}_i = -\sum_j \frac{k}{r_{ij}^2}\mathbf{n}_{ij} \qquad (1)$$

In (1), r_{ij} is the Euclidian distance between processes *i* and *j* in our architecture mapping, k is a constant, and $\mathbf{n}_{ij}$ is a unit vector pointing from process *i* to process *j* [40]. If the process is within range of the boundary of the architecture, the closest point on the boundary is treated as another single repulsive force on the process. The process moves one unit per time step in the direction of the net force. The length of the process's step is equal to the magnitude of the net force up to a maximum step size.

The algorithm exhibits the desirable emergent properties prevalent in related work in heat diffusion and robotic swarming algorithms. It shares attractive properties with heat diffusion in that it uses a local iterative scheme, requires no global communication, and provides global convergence for complex large-scale load distributions. However, to achieve resilience we add two additional process scheduling requirements: Process *replicas* must maintain locality and be

mapped to different processors. Recall that locality provides a mechanism upon which to gauge timeout durations for detecting inconsistent behavior between replicas. Mapping process replicas within a process group to different processors is a core requirement for resilience. We achieve process locality through the emergent property of swarm cohesion provided by robotic swarming techniques [35], [40]-[46].

In the DIFFUSE algorithm, a potential field is applied to the search domain corresponding to a map of processes in the architecture. Repulsive forces in the field are used to repel processes away from neighbors and obstacles, corresponding to the architecture boundary [40]. These repulsive forces are inversely proportional to the distance between processes and are limited in range. The emergent behaviors of the algorithm thus accomplish our core process scheduling requirements: processes diffuse across the architecture, maintain locality, and are mapped to different processors. The apparent diffusion of processes is a result of the virtual potential field dispersing processes uniformly throughout the system. Since repulsion weakens with distance and the process step size is limited, the swarm dynamics prevent processes from repelling too far and violating locality. Finally, strong repulsion at close range prevents processes from being scheduled to the same processor.

IV. Quantitative Analysis

Figure 4 illustrates the large scale benchmarks we use to compare competing algorithms [21]. This set expands on the core benchmark studies detailed in [20] through replication; details of the benchmarks are presented in the Appendix. We have designed the benchmarks to capture the primary attributes of the general process scheduling problem. The same basic static load distribution is considered on a two-dimensional mesh with three alternative initial process distributions. The light peaks in the benchmark are load troughs, or clusters of CPUs with

available resources. The load troughs vary in size (width) and intensity (amount of resources available). The characterization and distribution of the load troughs ensure that troughs are occluded by other troughs of lesser, greater, or equal intensity. Extensive dead space, representing fully loaded computers, is included to determine the impact of an imperceptible gradient in the amount of CPU resources available on search performance. Although a two-dimensional mesh is presented, the problem can be applied to any architecture through an appropriate virtual to physical machine translation.

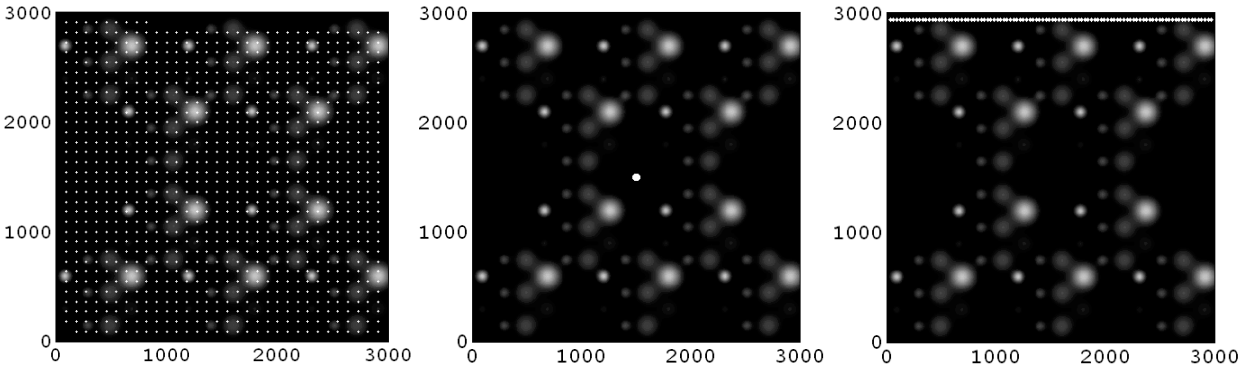
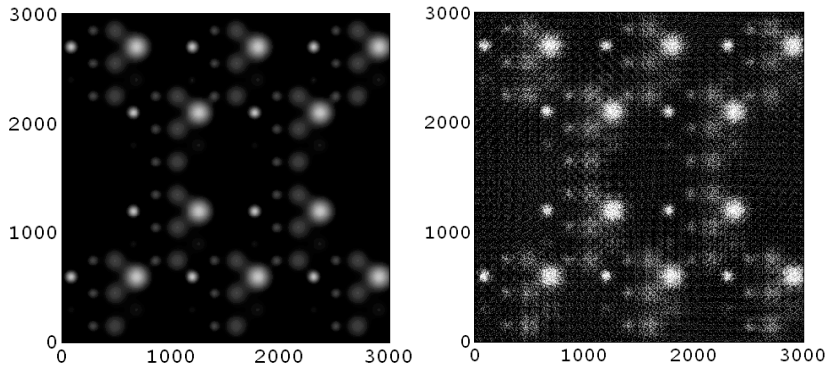


Fig. 4. (a) Uniform, (b) Point, and (c) Line initial distributions.

All three benchmarks contain the same load distribution consisting of 100 Gaussian load troughs on a 3000 x 3000 unit mesh. The field is constructed by replication of a parameterized group of 10 troughs that are 5 to 25 units in height and approximately 100 to 300 units in width. Dead space is created by setting the resource value to zero if it is below a threshold value of 0.5. Three initial process distributions are the *uniform*, *point*, and *line* distributions. The uniform distribution, common in the swarming literature [15], provides total coverage of the mesh and might correspond to a uniformly scattering process scheduler. The point distribution represents initial process deployment from a small cluster of CPUs in the center of the mesh. The line distribution corresponds to scheduling processes along one dimension of the architecture.

The benchmarks in Figure 4 feature a smoothly-varying gradient field that serves as an initial test case [21]. However, a successful process scheduler must be robust to random irregularities in load distribution. We introduce reproducible noise in the benchmark cases to assess the impact of irregular loads and resource gradients that are not smooth on load balancing performance. Pseudo-random noise is added to the original benchmark field by applying a reproducible, discrete *noise mask* throughout the mesh. First, a two-dimensional grid of 100 x 100 random numbers uniformly distributed in the range [-10, 10] was created. Then, this grid was replicated throughout the mesh and the corresponding random value was added to the each unit of the original benchmark field. Figure 5 displays the new gradient field that represents the noisy resource distribution in all three benchmark cases of our noise experiments. The original benchmark field is included for comparison. The detailed definition of our noise mask can be found in the Appendix.



(a) (b)

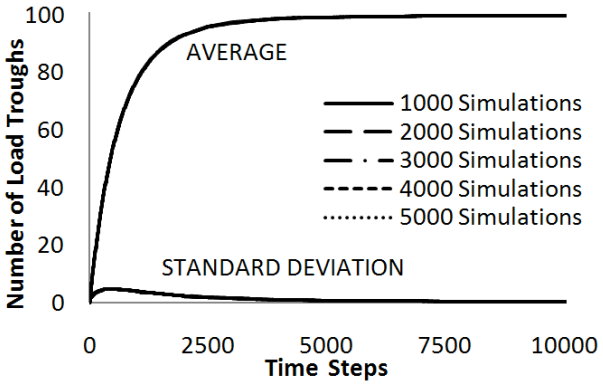
Fig. 5. (a) Original and (b) Noisy benchmark field.

The algorithms compared in this paper were evaluated through 1000 simulations on each benchmark case, unless otherwise specified. A swarm of 1000 processes is deployed. A process "finds" a load trough when it is scheduled within 10 units of the center of the load trough and remains on that load trough for the remainder of the search. Simulations are terminated if all

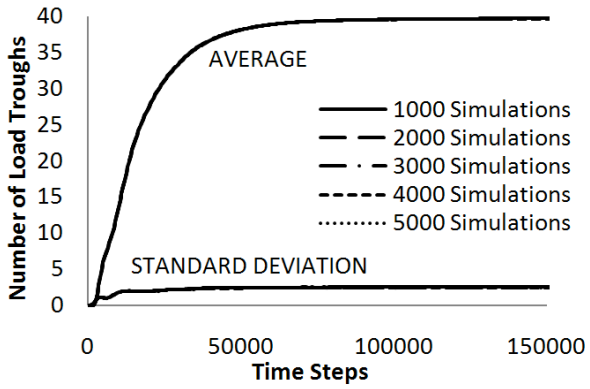
load troughs are utilized, if all processes are scheduled on a trough, or after 500,000 time steps. The time steps of the simulations account for each process movement and each BRW process tumble. All processes move at a velocity of one unit per time step, and a BRW tumble is conducted in one time step. The simulated communication range of the GSO, GSO/BRW hybrid, and DIFFUSE algorithms is 125 units.

The performance metrics of the benchmarks are the average number of load troughs found and the convergence times for each algorithm. The number of load troughs found at each time step is recorded for each simulation, and the average number of troughs found is calculated for 1000 simulations. The 95% confidence intervals on the mean are calculated from the sample mean and standard deviation of 1000 simulations to establish an error bar on the average performance. We define two convergence times to measure algorithm efficiency. The 75% convergence and 95% convergence metrics are the number of time steps to find an average of 75% and 95% of load troughs.

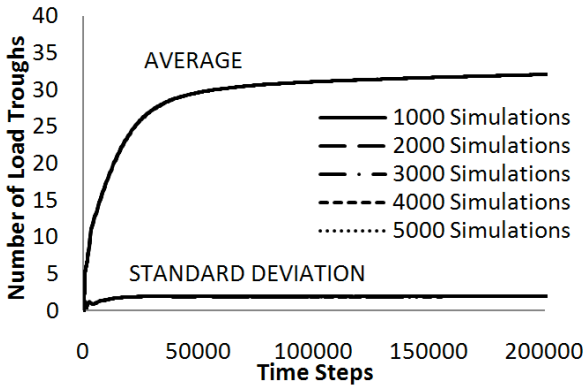
In order to compare the relative performance of the algorithms, it is necessary to determine the minimum number of simulations that yield reliable metrics. We conducted a sensitivity experiment to determine this number. A BRW was simulated 5000 times on each benchmark, and the mean and standard deviation of the number of load troughs found after 1000, 2000, 3000, 4000, and 5000 simulations were compared. The standard deviation and average number of troughs found at each time step for each benchmark are shown in Figure 6. In all three plots, the curves are too close to discern. Neither the average nor the standard deviation of troughs found change significantly for more than 1000 simulations. Therefore, 1000 simulations were sufficient for our performance comparisons.



(a)



(b)

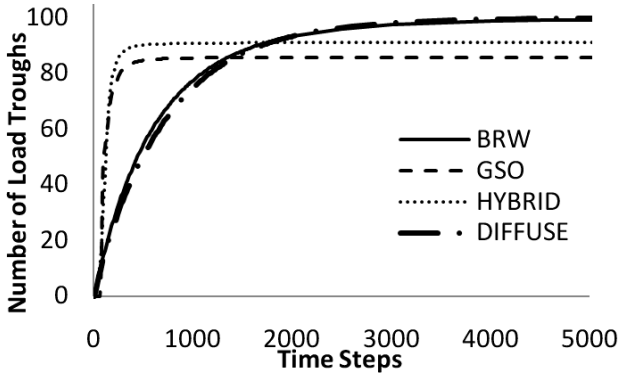


(c)

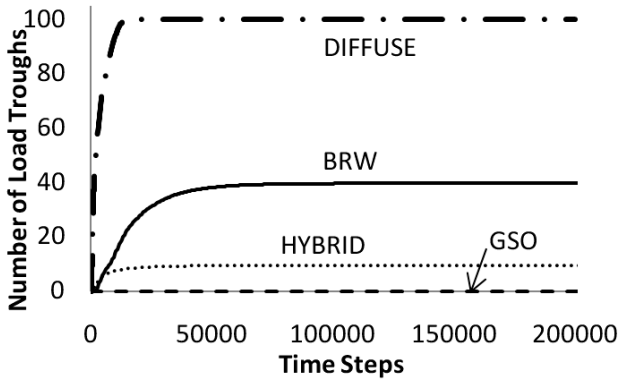
Fig. 6. Average load troughs found v. time step and standard deviation of load troughs found v. time step for BRW on the (a) uniform, (b) point, and (c) line initial distribution benchmark cases after 1000, 2000, 3000, 4000, and 5000 simulations.

Figure 7 shows the average number of load troughs found at each time step for 1000 simulations of the BRW, GSO, GSO/BRW HYBRID, and DIFFUSE algorithms. On the uniform benchmark, the DIFFUSE algorithm is the only algorithm that locates all 100 load troughs on average. The BRW algorithm performs satisfactorily, locating an average of 99.950 +/- 0.014 troughs with efficiency comparable to DIFFUSE. With uniform initial distribution, processes are dispersed throughout the architecture from the beginning. As a result, there is little advantage to the DISPERSE mode of our algorithm in this case.

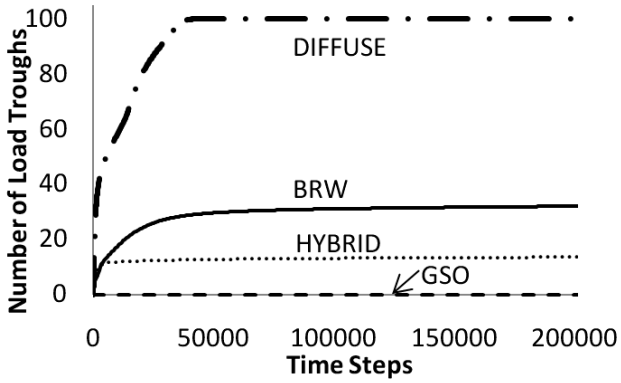
Neither the GSO nor the GSO/BRW HYBRID algorithm achieves 95% convergence on the uniform benchmark. This result is because the GSO algorithm favors load troughs with the most CPU resources available. Consider a GSO process with two neighbors, each located near a distinct load trough. The process is more likely to move toward the neighbor near the larger trough and overlook the trough with fewer resources. As a result, the smaller trough may not be located, leaving CPU resources unutilized. Conversely, the GSO and HYBRID algorithms achieve 75% convergence faster than the DIFFUSE algorithm. This outcome is due to the greediness of the GSO algorithm: the processes always move toward neighbors with greater CPU resources available. In contrast, the DISPERSE mode of our algorithm can slow localization when the swarm is already diffused by delaying processes from executing local search.



(a)



(b)



(c)

Fig. 7. Average load troughs found v. time step for BRW, DIFFUSE, GSO, and HYBRID on the (a) uniform, (b) point, and (c) line initial distribution benchmark cases.

The DIFFUSE algorithm locates all 100 load troughs on the point and line benchmark cases, while none of the other algorithms achieve 75% convergence on either case. The BRW

algorithm fails on the non-uniform benchmarks for two reasons. Some processes exit the benchmark boundaries because the BRW has no knowledge of the architecture domain. Also, multiple processes locate the same load trough. The BRW is an independent algorithm in which a process stops at the first trough it locates, even if that trough has already been found. The DIFFUSE algorithm uses the potential field to prevent processes from exiting the mesh boundary and to prevent multiple processes from converging on the same trough. These features enable the DIFFUSE algorithm to locate *all* load troughs.

The GSO algorithm locates zero sources in the point and line benchmarks. One reason for this failure is that GSO is handicapped by the dead space in the resource field. The point initial distribution deploys the entire swarm in the dead space, so all processes have equal luciferin. We learned in the initial benchmark study that GSO processes in this circumstance do not move. The BRW step in the HYBRID algorithm enables isolated processes to explore the architecture but only until the process locates a neighbor. Then, a strictly local search is initiated. Consequently, the swarm only locates load troughs that are nearby the initial deployment.

The DIFFUSE algorithm excels when the initial process distribution is non-uniform because the DISPERSE mode spreads processes across the architecture from the start. Subsequently, load troughs that are distant from the initial process deployment are located quickly. The DIFFUSE algorithm outperforms the other algorithms because the potential field forces one of the crucial measures that we identified in the initial benchmark experiments: broad exploration of the entire architecture.

Table I presents a summary of the performance metrics of the DIFFUSE, BRW, GSO, and HYBRID algorithms. The table includes the number of time steps for 75% and 95% convergence and the total average number of load troughs found for each algorithm. The

algorithms are ranked by the average load troughs found total. The dashes indicate that the algorithm did not converge.

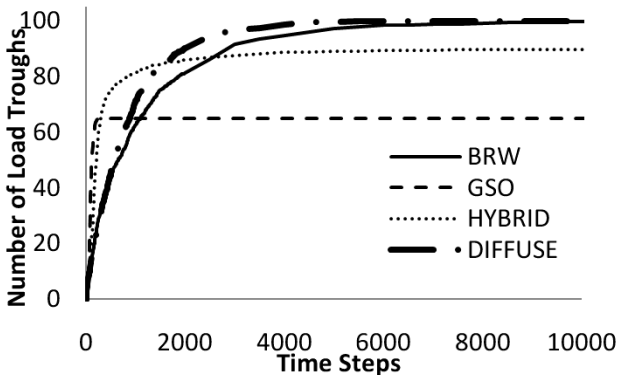
TABLE I

Summary of Algorithm Performance

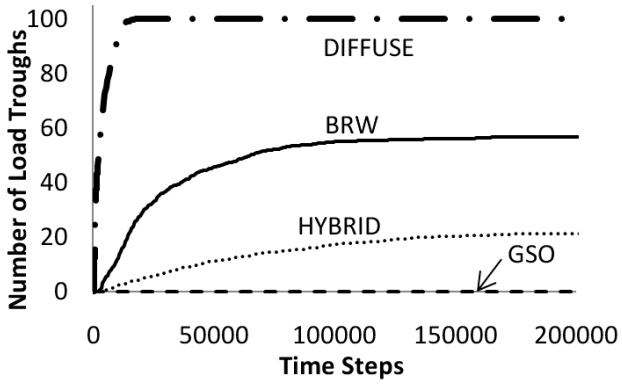
	Algorithm	75% Convergence Time Steps	95% Convergence Time Steps	Average Load Troughs Found Total
Uniform	DIFFUSE	1010	2500	100.000 +/- 0.000
	BRW	930	2500	99.950 +/- 0.014
	HYBRID	180	-	91.674 +/- 0.104
	GSO	210	-	85.922 +/- 0.110
Point	DIFFUSE	5400	10800	100.000 +/- 0.000
	BRW	-	-	39.758 +/- 0.158
	HYBRID	-	-	9.344 +/- 0.092
	GSO	-	-	0.000 +/- 0.000
Line	DIFFUSE	18000	32600	100.000 +/- 0.000
	BRW	-	-	33.750 +/- 0.132
	HYBRID	-	-	14.148 +/- 0.085
	GSO	-	-	0.000 +/- 0.000

To evaluate the impact of noisy resource gradients on search performance, we simulated 10 searches of each algorithm on the noisy uniform, point, and line benchmark cases. The simulation conditions and performance metrics were the same as the original benchmark experiments. The average number of load troughs found at each time step for each benchmark case is plotted in Figure 8. The DIFFUSE algorithm exhibits the best overall performance of all four algorithms. It is the only algorithm that locates all 100 load troughs in all three benchmarks. In fact, the DIFFUSE algorithm performance here is very similar to the performance on the

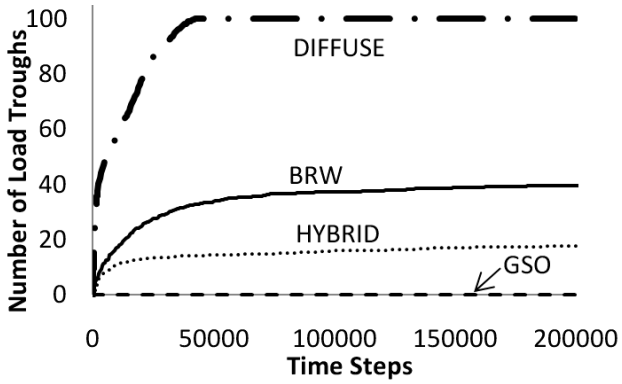
benchmarks without noise. We believe this similarity is because process diffusion is independent of the resource distribution and based only on limited range interactions between processes. The noise does not hinder the diffusion of processes, which accelerates load trough localization for the DIFFUSE algorithm.



(a)



(b)



(c)

Fig. 8. Average load troughs found v. time step for BRW, DIFFUSE, GSO, and HYBRID on the (a) uniform, (b) point, and (c) line initial distribution noisy benchmark cases.

The other algorithms are more affected by the noise in the benchmarks than our DIFFUSE algorithm. The BRW algorithm, as well as the BRW component of the HYBRID algorithm, performs better in the presence of noise. We believe the noise emphasizes the random nature of the BRW, which encourages processes to explore the architecture. Processes may be less susceptible to being drawn to the first trough they encounter. The GSO algorithm locates fewer sources in the uniform benchmark and, surprisingly, zero sources in the point and line benchmarks. We attribute this result to the fact that some noise mimics local maxima in the benchmark field. A GSO process can be scheduled on a CPU with a large noise contribution and attract processes within range. In this way, subgroups of the swarm can be trapped on a cluster with few under-utilized processors. These subgroups are effectively removed from cooperative search and degrade the algorithm's overall search performance.

V. Summary and Conclusions

The goals of this research are to improve the resilience of mission-critical applications to a wide variety of failures, errors, and malicious attacks. Our approach is to dynamically replicate processes, detect inconsistencies in their behavior, and transparently restore the level of fault tolerance as a computation proceeds. This paper describes a concurrent process scheduling algorithm that manages replicated processes, inspired by the notions of heat diffusion and robotic swarming. Heat diffusion is emulated to disseminate processes across computer architecture. Robotic swarming techniques are used to maintain locality between replicated processes while balancing load. Comparative analysis, between the algorithm and three alternatives taken from the robotics literature, was conducted using a set of large scale benchmark cases that feature 100 load troughs on large scale mesh architecture containing 9,000,000 computers.

The DIFFUSE algorithm locates all load troughs on all benchmark cases, unlike the other algorithms. In particular, it outperforms the other algorithms when the initial process distribution is non-uniform. This success is accomplished by diffusing processes across the architecture using only limited range communication. In addition, swarm-based obstacle avoidance techniques prevent multiple processes from converging on the same load trough and prevent processes from exiting the computer domain. Together, these features ensure that the DIFFUSE algorithm to locates *all* load troughs in the benchmarks.

The algorithms were evaluated on a second benchmark set with noisy load distributions representing irregular load statistics. The DIFFUSE algorithm proves to be robust to noise; the dispersion of processes is dominated by swarm dynamics rather than the local load statistics. Consequently, the swarm as a whole is not susceptible to local maxima or isolated irregularities in the load distribution. Individually, diffused processes use load statistics to search *locally* for unloaded processors and optimize utilization overall.

Swarm-inspired process scheduling is relatively new and may offer alternative solutions to a wide range of problems in distributed resource management. As multi-core computers become ubiquitous, we see an increase in number and diversity of devices and communication networks available as computational engines. Distributed systems are now interconnected through a wide variety of local area, wide area, and wireless networking technologies. This diversity confounds the problem of distributed resource management and the notion of *local* communication. The approaches described in this paper include a spectrum of communication ranges for distributed scheduling. The BRW algorithm is an autonomous algorithm with no communication. This autonomy could be applied to any computer architecture. The original heat diffusion algorithm conducts only nearest neighbor communication. This is a natural communication scheme for

large scale architectures with regular grid connectivity, such as mesh architecture. The GSO and DIFFUSE algorithms represent an intermediate communication class between nearest neighbor and global communication featuring limited range communication. These algorithms may be applicable to a wider variety of communications networks. For example, in local area networks, locality may be defined by the number of network hops whereas in wireless networks, locality may be defined by a physical distance. Further, limited range communication algorithms may provide the potential to tune the communication range to a specific platform or network. This versatility is a potential advantage of the DIFFUSE algorithm and other algorithms in this class.

This paper extends and consolidates a progression of research to explore robotic swarming algorithms in the context of resiliency. Initially, we devised a set of benchmark cases to capture the core scheduling problem [20]. This benchmark set was extended to represent large scale architectures [21]. We evaluated the performance of algorithms from the robotics literature on both sets to assess their relative performance, but none of the robotics algorithms succeeded on all of the benchmarks [20], [21]. The DIFFUSE algorithm presented here improves on the benchmark performance of the earlier studies and continues to provide superior performance when operating in the presence of noise.

As we proceed, we will continue to optimize the algorithm for management of multiple concurrent applications, similar to our previous work in [14], [23]. By exploring process interactions, we hope to optimize the tradeoffs between maintaining process *locality* for reduced transit delays and process *separation* for resilience.

References

[1] D.A. Patterson, G. Gibson, and R.H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management Data*, 1988, pp. 109-116.

[2] Lefurgy, X. Wang, and M. Ware, "Server-level power control," in *Proceedings of the International Conference on Autonomic Computing*, 2007, pp. 4-13.

[3] H. Debar, M. Dacier, and A. Wespi. "A revised taxonomy for intrusion-detection systems," *Annals of Telecommunications*, vol. 55, no. 7, pp. 361-378, 2000.

[4] R. Di Pietro and L.V. Mancini, *Intrusion Detection Systems*, New York, Springer-Verlag, 2008.

[5] Singhal, "Intrusion Detection Systems," in *Data Warehousing and Data Mining for Cyber Security*, vol. 31, pp. 43-47, 2007.

[6] G.H. Kim and E.H. Spafford, "The design and implementation of tripwire: A file system integrity checker," in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994, pp. 18-29.

[7] J. Kaczmarek and M. Wrobel, "Modern approaches to file system integrity checking," in *Proceedings of the 1st International Conference on Information Technology*, 2008.

[8] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proceedings of the Network and Distributed Systems Security Symposium*, 2003, pp. 191-206.

[9] N.L. Petroni, T. Fraser, J. Molina, and W.A. Arbaugh, "Copilot- a Coprocessor-based Kernel Runtime Integrity Monitor," in *Proceedings of the 13th USENIX Security Symposium*, 2004, pp. 179-194.

[10] N.A.Quynh and Y. Takefuji, “Towards a tamper-resistant kernel rootkit detector,” in *Proceedings of the 2007 ACM Symposium on Applied Computing*, 2007, pp. 276-283.

[11] R. Riley, X. Jiang, and D. Xu, “Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing,” in *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, 2008, pp. 1-20.

[12] J. Lee., S.J. Chapin, and S. Taylor. "Computational Resiliency", *Journal of Quality and Reliability Engineering International,* vol. 18, no. 3, pp 185-199, 2002.

[13] G. Cybenko, “Dynamic load balancing for distributed memory multiprocessors,” *Journal of Parallel and Distributed Computing*, vol. 7, no. 2, pp. 279-301, 1989.

[14] Heirich and S. Taylor "Load Balancing by Diffusion", in *Proceedings of 24th International Conference on Parallel Programming*, 1995, pp. 192-202.

[15] Dhariwal, G.S. Suhkatme, and A.A.G. Requicha, “Bacterium-inspired robots for environmental monitoring,” in *Proceedings of the 2004 International conference on Robotics & Automation*, New Orleans, Louisiana, April 2004, pp. 1436-1443.

[16] K.N. Krishnanand and D. Ghose, “A glowworm swarm optimization based multi-robot system for signal source localization,” in *Design and Control of Intelligent Robotic Systems,* D. Liu, L. Wang, and K.C. Tan, Eds., Berlin, Germany, Springer Verlag, 2009, pp. 49-68.

[17] K.N. Krishnanand and D. Ghose, “Glowworm swarm optimization for simultaneous capture of multiple local optima of multimodal functions,” *Swarm Intelligence*, vol.3, no. 2, pp.87-124, 2009.

[18] K.N. Krishnanand and D. Ghose, "Glowworm swarm based optimization algorithm for multimodal functions with collective robotics applications," *Multiagent and Grid Systems,* vol. 2, no. 3, pp.209-222, 2006.

[19] K.N. Krishnanand and D. Ghose, "Glowworm swarm optimization: A swarm intelligence based multimodal function optimization technique with applications to multiple signal source localization," Technical Report GCDSL 2007/05, Department of Aerospace Engineering, Indian Institute of Science, October 2007.

[20] K. McGill and S. Taylor, "Comparing swarm algorithms for multi-source localization," In *Proceedings of the 2009 IEEE International Workshop on Safety, Security, and Rescue Robotics*, Denver, Colorado, November 3-6, 2009.

[21] K. McGill and S. Taylor, "Comparing swarm algorithms for large scale multi-source localization," In *Proceedings of the 2009 IEEE International Conference on Technologies for Practical Robot Applications*, Woburn, Massachusetts, November 9-10, 2009.

[22] K. McGill and S. Taylor, "Robot Algorithms for Localization of Multiple Emission Sources," ACM *Computing Surveys (CSUR)*, accepted for publication.

[23] A.B. Heirich, "Analysis of scalable algorithms for dynamic load balancing and mapping with application to photo-realistic rendering," Ph.D. dissertation, California Institute of Technology, Pasadena, CA, 1998.

[24] J. E. Boillat, "Load balancing and poisson equation in a graph," *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 289-313, 1990.

[25] P. Berenbrink, T. Friedetzky, and Z. Hu, "A new analytical method for parallel, diffusion-type load balancing," *Journal of Parallel and Distributed Computing,*vol. 69, pp. 54-61, 2009.

[26] E. Jeannot and F. Vernier, "A practical approach of diffusive load balancing algorithms," in *Euro-Par 2006 Parallel Processing*, W. Nagel, W. Walter, and W. Lehner, Eds., Springer Verlag, 2006.

[27] T. Rotaru and H. Nageli, "Dynamic load balancing by diffusion in heterogeneous systems," *Journal of Parallel and Distributed Computing*, vol. 64, no.4, pp. 481-497, 2004.

[28] J. Watts, "A Practical Approach to Dynamic Load Balancing," M.S. Thesis, California Institute of Technology, Pasadena, CA, 1995.

[29] Xu, B. Monien, R. Luling, and F.C.M.Lau, "Nearest neighbor algorithms for load balancing in parallel computers" *Concurrency: Practice and Experience*, vol. 7, no. 7, pp. 707-736, 1995.

[30] S.S. Hosseini, B. Litow, M. Malkawi, J. McPherson, and K. Vairavan, "Analysis of a graph coloring based distributed load balancing algorithm," *Journal of Parallel and Distributed Computing*, vol. 10, no. 2, pp. 160-166, 1990.

[31] M. Houle, A. Symvonis, and D.R. Wood, "Dimension-exchange algorithms for load balancing on trees," in *Proceedings of the 9th International Colloquium on Structure Information and Communication Complexity*, 2002, pp. 181-196.

[32] G. Bronevich and W. Meyer, "Load balancing algorithms based on gradient methods and their analysis through algebraic graph theory," *Journal of Parallel and Distributed Computing*, vol. 68, no. 2, pp. 209-220, 2008.

[33] F.C.H. Lin and R.M. Keller, "The gradient model load balancing method," *IEEE Transactions on Software Engineering*, vol. 13, no. 1, pp. 32-28, 1987.

[34] X. Cui, C.T. Hardin, R.K. Ragade, and A.S. Elmaghraby, "A swarm approach for emission sources localization," in *Proceedings of the 16th International Conference on Tools with Artificial Intelligence*, Boca Raton, Florida, Nov 2004, pp. 424-430.

[35] V. Gazi and K. Passino, "Stability analysis of social foraging swarms," *IEEE Transactions on Systems, Man, and Cybernetics—Part B: cybernetics,* vol. 34, no. 1, pp. 539-557, 2004.

[36] L. Marques, U. Nunes, and A.T. De Almeida, "Odour searching with autonomous mobile robots: an evolutionary-based approach," in *Proceedings of the IEEE Int. Conf. on Advanced Robotics*, Combra, Portugal, June 2003, pp. 494-500.

[37] L. Marques, U. Nunes, and A.T. De Almeida, "Particle swarm-based olfactory guided search," *Autonomous Robots,* vol. 20, pp. 277-287, 2006.

[38] J. Pugh, and A. Martinoli, "Distributed Adaptation in multi-robot search using particle swarm optimization," in *Proceedings of the 10th International Conference on the Simulation of Adaptive Behavior*, Osaka, Japan, July 2008, pp. 393- 402.

[39] P. Scerri, T. Von Gonten, G. Fudge, S. Owens, and K. Sycara SCERRI, P., "Transitioning multiagent technology to UAV applications," in *Proceedings of 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, Estoril, Portugal, May 2008, pp. 89-96.

[40] A. Howard, M. Mataric, and G. Sukhatme, "Mobile sensor network deployment using potential fields: A distributed, scalable solution to the area coverage problem," in *Proceedings of the 6th International symposium on Distributed Autonomous Robotics Systems (DARSO2),* Fukuoka, Japan, June 2002, pp. 299-208.

[41] H. Wang and Y. Guo, "A decentralized control for mobile sensor network effective coverage," in *Proceedings of the 7th World Congress on Intelligent Control and Automation,* Chongqing, China, June 2008, pp. 473- 478.

[42] J. Cortes, S. Martinez, T. Karatax, and F. Bullo, "Coverage control for mobile sensing networks," *IEEE Transaction on Robotics and Automation*, vol. 20, no. 2, pp. 243- 255, April 2004.

[43] J. Cortes, S. Martinez, and F. Bullo, "Spatially-distributed coverage optimization and control with limited-range interactions," *ESAIM: Control, Optimisation and Calculus of Variations,* vol. 11, no. 4, pp. 691-719, October 2005.

[44] P. Ogren, E. Fiorelli, and N.E. Leonard, "Cooperative control of mobile sensor networks: adaptive gradient climbing in a distributed environment," *IEEE Transactions on Automatic Control*, vol. 49, no. 8, pp. 1292-1302, August 2004.

[45] D. Zarzhitsky, D. Spears, and W. Spears, "Swarms for chemical plume tracing," in *Proceedings of the IEEE Swarm Intelligence Symposium (SIS'05)*, Pasadena, California, June 2005, pp. 249-256.

[46] W. Spears, D. Spears, J. Hamann, and R. Heil, "Distributed, physics-based control of swarms of vehicles," *Autonomous Robots*, vol. 17, no. 2-3, pp. 137-162, September 2004.

[47] K. McGill, "McGill Research," 9/15/2010, Available at: http://www.engineering.dartmouth.edu/~Kathleen_N_McGill/research.html.

Appendix

All three benchmark cases use the same gradient field consisting of 100 Gaussian sources. The field strength from all the sources at any point (x, y) in the field is a summation of the components from the individual sources given by (2).

$$f(x,y) = \sum_{i=1}^{n} I_i e^{-\frac{((x-x_i)^2+(y-y_i)^2)}{2\sigma_i^2}} \qquad (2)$$

In (2), the *i*th source is located at point (x_i, y_i) in the search space and has intensity I_i and "width" σ_i. After calculating the total field strength from all sources, any field strength less than a threshold of 0.5 is set to zero.

The field is constructed by replicating a parameterized group of 10 sources. Four parameters are required to characterize each source according to (2). The parameters for these sources are shown in Table II. Table II serves as a 10 source template to replicate in the space. The x_i and y_i values in the table are *relative coordinates* with respect to the position of the first source. The *absolute* position of the first source of each replication is shown in Table III. The coordinates in Table III provide the locations to replicate each 10 source template.

TABLE II

Source Parameters

Source *i*	x_i	y_i	I_i	σ_i
1	0	0	25	35
2	0	-300	5	35
3	200	150	10	35
4	200	-150	10	35
5	200	-450	10	35
6	400	150	10	75
7	400	-150	10	75
8	400	-450	10	75
9	600	0	25	75
10	600	-300	5	75

TABLE III

Absolute Position of the Source Replications

Replication	x_1	y_1
1	90	600
2	1200	600
3	2310	600
4	660	2100
5	1770	2100
6	660	1200
7	1770	1200
8	90	2700
9	1200	2700
10	2310	2700

We added pseudo-random noise to the original benchmark field by applying a discrete *noise mask* throughout the mesh. We generated a grid of 100 x 100 random numbers uniformly distributed in the range [-10, 10] using Microsoft Excel. The table of random numbers is too large to print here but can be accessed online [47]. We replicate the noise mask throughout the mesh by using the grid as a lookup table and adding the corresponding random value to the each unit of the original benchmark field. To determine the appropriate table indexes for the random number lookup, we perform modular arithmetic using the x- and y-locations of the processor in the mesh (x mod 100, y mod 100). For example, the random number that is added to the processor at position (373, 1988) in the mesh is at position (73, 88) in the lookup table.