Control Points for Adaptive Parallel Performance Tuning

Isaac Dooley [*], Laxmikant V. Kale
University of Illinois
Department of Computer Science

November 3, 2008

Abstract

This paper describes a new parallel program tuning framework, with a new approach for tuning. The application exposes tuning parameters while an adaptive runtime system tunes the values of the parameters as the program executes. The parameters are allowed to vary throughout the duration of a run of a program. This provides two benefits over using a static set of parameters for each program run. The first is that parameters affecting performance may need to vary throughout the program run to achieve maximum performance. The second benefit is that many parameter configurations can be tested in a single run of an application. This approach amortizes the cost of startup across many evaluations of configurations in the parameter space.

The framework can tune parameters across multiple modules in a parallel application. It can collectively optimize the parameters, called *Control Points*, exposed by the application, the runtime system, and libraries. This paper describes preliminary work using Control Points with two programs, a finite difference scheme, and a pipelined filtering application. Results from an exhaustive search of the configuration space are provided as a basis for determining which types of search strategies would be effective. The optimal configurations for control points vary between different parallel systems.

1 Control Points

Control points are parameters or knobs that control aspects of a parallel program's behavior. They can be used to choose between multiple algorithms, or to determine the block sizes for a parallel data decomposition. Control points are exposed in a parallel program or library wherever a choice can be made between alternative implementations or variants of an algorithm. The actual choice of control point values is performed by a framework in the parallel runtime system, using previously collected performance data to guide a search for an optimal configuration. Collected performance data associated with control point configurations are saved in a database on disk for use in subsequent parallel executions. Various optimization schemes can be used to choose a good configuration to use. The term *Control Points* comes from [1].

The traditional programming model used in adaptive runtime systems such as Charm++ is one where the programmer specifies a collection of interacting parallel objects or tasks [2]. The job of mapping these objects to available resources is done by the runtime system, not the programmer. The benefits of such a programming model is that some of the parallel programming burden is eliminated and the runtime system can observe and adapt the program to the available parallel system by performing dynamic load balancing or communication optimizations. Although this approach has worked well when scaling some scientific applications to over 40,000 processors, the system does not provide any mechanisms for feedback to the application from the parallel runtime system. The new work described in this paper attempts to show that applications can perform better with some simple extensions allowing the runtime system to feed information back to and exercise control over the application. This information could be based upon observed behaviors such as observed communication patterns, memory usage patterns, or other performance characteristics. The runtime system will synthesize such information with the instrumented values of control points to determine optimal control point configurations.

[*]This work was funded in part by the DOE High Performance Computer Science Fellowship Program.

It has previously been shown that a runtime system or communication library can benefit from effectively choosing between multiple algorithms or protocols for a single communication pattern [3] [4], [5]. Different algorithms and protocols have performance tradeoffs involving latencies, latency tolerance, total communication bandwidth, usefulness on different network topologies, or transient memory overheads. Similarly, numerical libraries such as FFTW or ATLAS instrument and choose between multiple versions of an algorithm. Existing systems switch between different algorithms dynamically, however they do not expose hints for the application to modify its behavior.

Control points can be used to expose tunable parameters across multiple modules or libraries that make up the complicated parallel programs in use today. Control points can be exposed both from within an application's code as well as in parallel libraries or even within the runtime system itself. The application programmer exposes some of these control points, while the communication library exposes control points and the numerical libraries expose other control points. When the control points are exposed from all these different modules in a program, these control point knobs can be co-optimized.

Simple hooks for an adaptive runtime system to instruct an application to modify its behavior can provide a new space for previously unavailable application optimizations. Control points are the mechanism proposed in this paper to provide feedback from a runtime system to an application, and to provide a space over which performance can be optimized.

2 Control Point Framework API

An application requests the value for each control point by calling a simple function `controlPoint`. This function takes as parameters the name of the control point, and the range of acceptable integer values for the control point. The function returns the control point value which is consistent across all processors within a phase. A phase is a time range during which a single configuration of control point values is returned. The application can explicitly advance to the next phase when required, or the phase can be automatically advanced periodically.

```
int controlPointValue = controlPoint("Control Point Name", 1, 50);
```

Alternate versions of the function are provide for convenience. These provide similar functionality, but take as a parameter a list containing the range of possible integer values, or return only integral powers of two within the range.

For the preliminary control point framework implementation, the timings representing the application performance are provided by the application. It is trivial for a tightly syncrhonized application to provide time values for each timestep. All the provided times are recorded by the framework in a function call that simply takes a `time` parameter:

```
registerControlPointTiming(time);
```

The control point framework is unique because it attempts to feed information back to the application. Its mechanism for feedback is a callback to the application. The callback is invoked by the runtime system periodically. The callback is a standard Charm++ callback provided at startup by the application through a registration call such as

```
CkCallback myCallback (CkIndex_Main::controlPointChange(NULL),proxy);
registerControlPointChangeCallback(myCallback);
```

In the callback method, the application simply gets the new updated control point values for the phase, and adjusts its behavior appropriately. For example, the program could instruct its data array library to repartition the data arrays using a new granularity, or it could switch to the specified algorithm for all future uses during the phase.

Future plans for the API include adding mechanisms by which the programmer can associate a high level meaning to each control point's values. For example, information could be conveyed to the control point framework so that the control point framework could know which direction to adjust a control point to increase or decrease available concurrency. Alternatively, a control point could increase or decrease total communication volume by using different numbers of phases for an all-to-all communication operation. The runtime system could observe the effects of varying control point values, but information provided by the programmer could be useful when efficiently optimizing the control point values. Once the runtime system has knowledge of the effects of modifying each control point, the search for optimal values can be more effectively guided, eliminating the need for a blind search for the optimal values.

3 Case studies

To verify the usefulness of the preliminary control point framework and to determine the suitability of optimization search strategies for the control point performance space, two initial parallel programs were modified to use the control point framework. The first program streams a pool of data through a pipelined filter. The second program implements a Finite Differencing scheme for the 2-D wave equation over a structured grid. Each program uses two control points, hence producing a 2-D space over which the performance can be optimized. The entire optimization space was exhaustively examined by the framework to provide an insight into the types of optimization schemes that could efficiently optimize the performance. This section displays the results from the exhaustive searches, each producing a different structure for the optimal regions of the control point space.

3.1 Case Study: Pipelined Stream Filtering

The first program streams a pool of data items through a pipeline of filter operations. Each stage in the pipeline contains a fraction of the entire filter pipeline. Specifically, the input pool of data is a set of 2000 rectangles which are filtered out of the set if they collide with any rectangle in a different set of 3000 rectangles. Each rectangle is axis-aligned and is represented by 4 double precision values.

The input data pool of 2000 items is chopped into slices that are fed sequentially into the pipeline. For all cases, the input pool is sliced into chunks containing C items, with the final slice possibly containing fewer items. C is selected from the set $\{1, 2, 4, \ldots, 1024\}$. Each chunk is fed into the pipeline in a separate message. After the entire message has been processed by a pipeline stage, a new message containing the filtered data is sent to the next pipeline stage. The number of pipeline stages, P, comes from the set $\{1, 2, 3, \ldots 64\}$. Figure 1 displays the parallel decomposition of the pipeline for two different sets of control point values. The two control points for the program are the sizes of the slices for the input data, and the number of pipeline stages. The problem size, including both the size of the data pool and the number of pipeline filter operations, remains fixed.

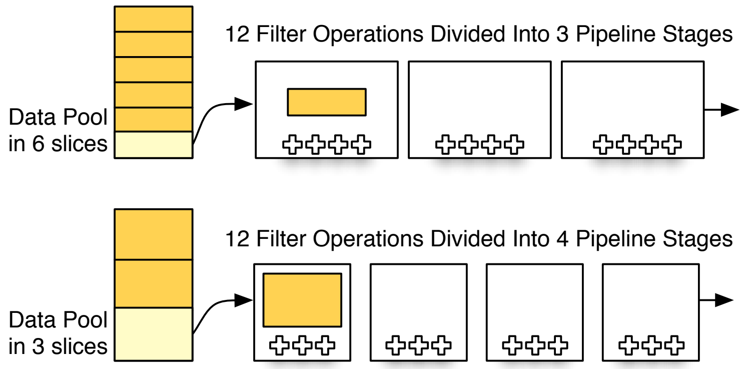
12 Filter Operations Divided Into 3 Pipeline Stages
Data Pool
in 6 slices
12 Filter Operations Divided Into 4 Pipeline Stages
Data Pool
in 3 slices

Figure 1: Two possible parallel decompositions of the Pipelined Stream Filtering Program.

Figure 2 shows the instrumented performance associated with all 640 points in the entire space of control point values. These instrumented results are for a run on 14 processors on 2 nodes of the NCSA Abe cluster. The performance for this program is easily explained. The cases for the small slices of the input data (top of Figure 2) have the lowest latency for the first item to flow through the entire pipeline, but the many total sent messages leads to excessive overhead. On the other hand, the cases for large slices (bottom of Figure 2) limit the available concurrency and increase the latency for the first data to pass through the entire pipeline. The decomposition of the pipeline also affects the performance of the program. If a small number of pipeline stages are used (left of Figure 2), only a subset of the available processors on the cluster are used. If a large number of pipeline stages are used (right of Figure 2), the over-decomposition creates extra overheads. Although the performance impacts for the control point values in this program is intuitive and simple, our second simple case study shows much more complicated performance variation across the control point space.

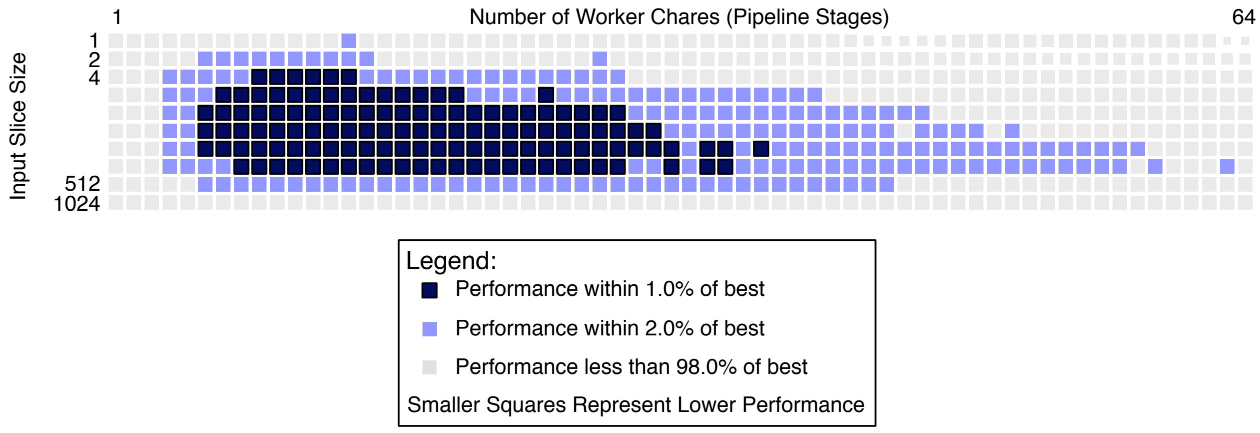

Figure 2: Performance for all possible parameter space configurations for the Pipelined Stream Filtering Program run on 14 processors (2 nodes) of the NCSA Abe cluster.

3.2 Case Study: Wave2D

The second case study program is a 2-D wave equation discretization on a 2-D rectangular grid. The program maintains two data arrays, one for a current timestep p^t, and one for a previous timestep p^{t-1}. The values in the array p^{t+1} for the next timestep are computed using the update rule,

$$p^{t+1}_{x,y} = c^2 \left(p^t_{x+1,y} + p^t_{x-1,y} + p^t_{x,y+1} + p^t_{x,y-1} - 4p^t_{x,y}\right) - p^{t-1}_{x,y} + 2p^t_{x,y}.$$

The program exposes two control points. The control points specify the number of worker objects into which each dimension of the 2-D data grid p is decomposed. A library external to the application provides support for dynamically repartitioning the 2-D grids. The coarsest decomposition uses just a single worker object that owns and updates the entire grid. The associated control point values for the coarsest decomposition is $(1, 1)$. The finest decomposition distributes the grid across a 50×50 array of worker objects. The control point values for this finest decomposition is $(50, 50)$ In all cases, the worker chare objects are round-robin mapped to the available processors in order to ensure an even load balance.

To evaluate the types of search strategies that are useful for this problem, an exhaustive search is performed over all 2500 configurations in the control point parameter space $\forall\,(x, y)\,, 1 \le x \le 50, 1 \le y \le 50$. This exhaustive search was performed on two test platforms. The first platform is 32 processors across 4 nodes in the NCSA Abe cluster. The second test platform is 6 processors of a single 8-core workstation. A 2-D data grid of size 1000×1000 was used on both platforms, and a 4000×4000 grid was also used on the latter platform. Figures 3, 4, and 5 show the instrumented performance results from these three exhaustive searches. The resulting performance plots over the parameter spaces are quite different between the two platforms, in potentially non-obvious ways. Furthermore, interesting unexpected patterns arise.

Figure 3 and 4 show that the optimal decomposition of the 2-D grids occur in a wide range of values, with asymmetries in the aspect ratios of the resulting grid partitions. The test case used in figure 3 shows that optimal configurations include large numbers of X or Y partitions, while figure 4 reveals that no configurations with large numbers of Y partitions perform optimally. The optimal configurations are similar, but distinct, when varying the array size on a single parallel system, as shown in figure 4 versus figure 5. Here, with a larger grid domain, fewer optimal configurations exist, and the optimal decomposition are all slices along the Y dimension. Although it is likely that cache effects are the cause of the significant portion of the variation in performance, the approach taken in this work does not require the programmer to reason about the precise effect of cache sizes or data layout on their program for each parallel system.

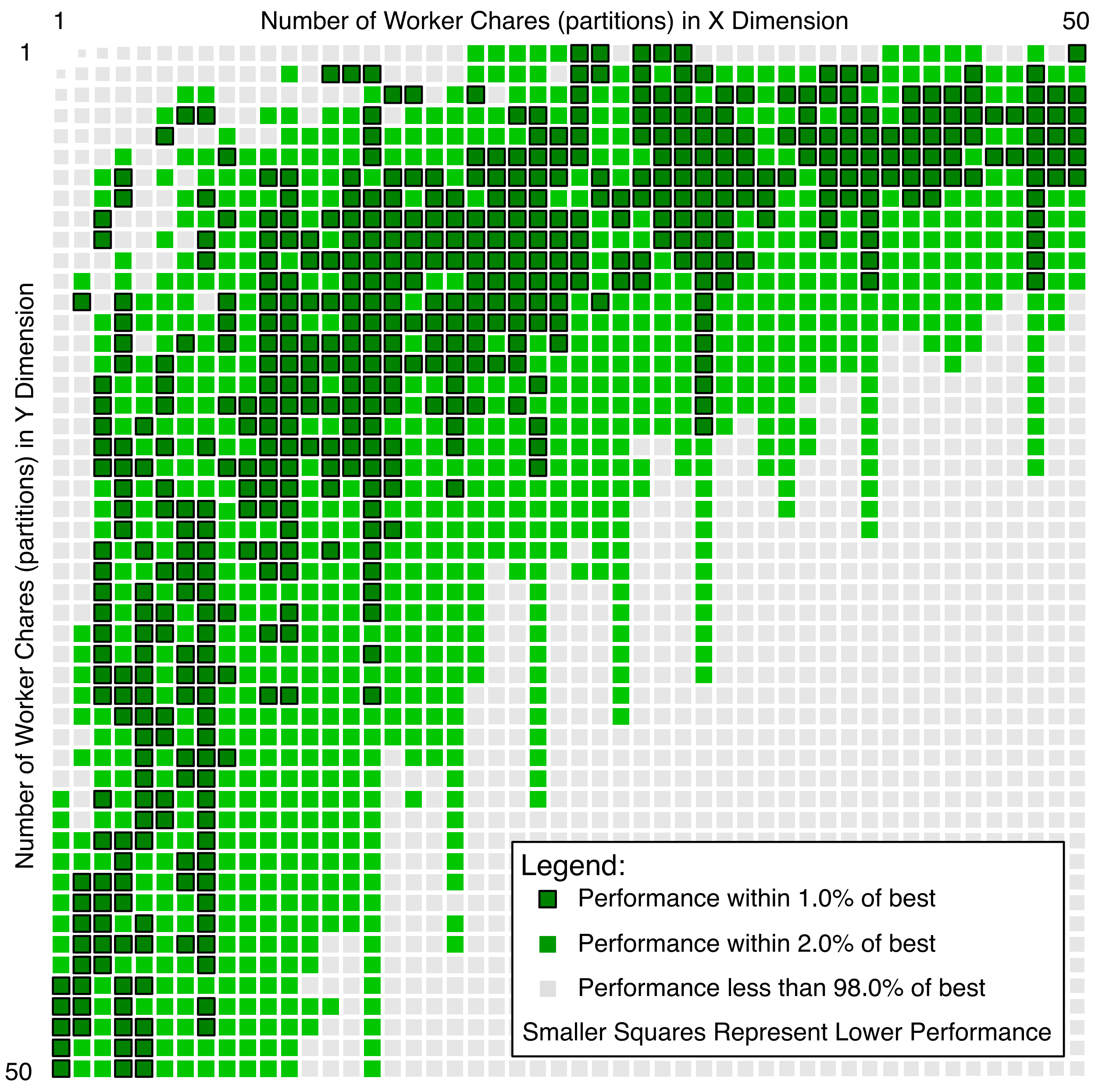

Figure 3: Performance for all possible parameter space configurations for the Wave2D program on a 1000×1000 data grid on 32 processors (4 nodes) of the NCSA Abe cluster. The performance for the upper left corner is low because the degree of available parallelism is too small. In the bottom right corner, the performance suffers because the problem is excessively over-decomposed causing significant overheads to limit scalability. Many close to optimal configurations exist! The optimal decompositions are blocks with varying aspect ratios, sometimes slices.

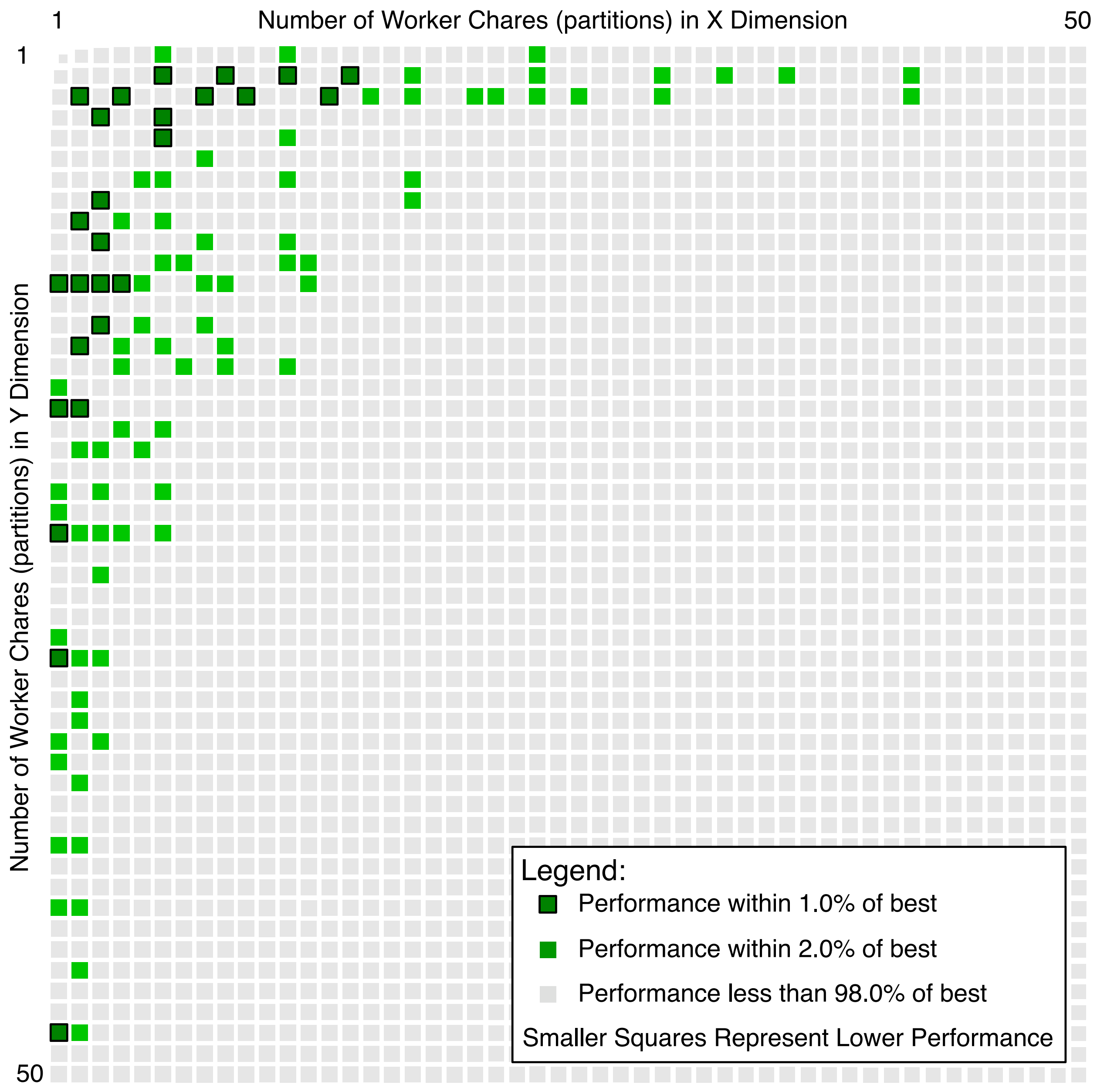

Figure 4: Performance for all possible parameter space configurations for the Wave2D program on a 1000×1000 data grid on 6 processors of an 8-core Intel Linux workstation. In this case, only a few of the tested configurations are close to optimal. The best performing decompositions of the 2-D grid data are scattered throughout the configuration space, with optimal values occurring at nicely load balanced cases where the number of total worker chares is a multiple of 6.

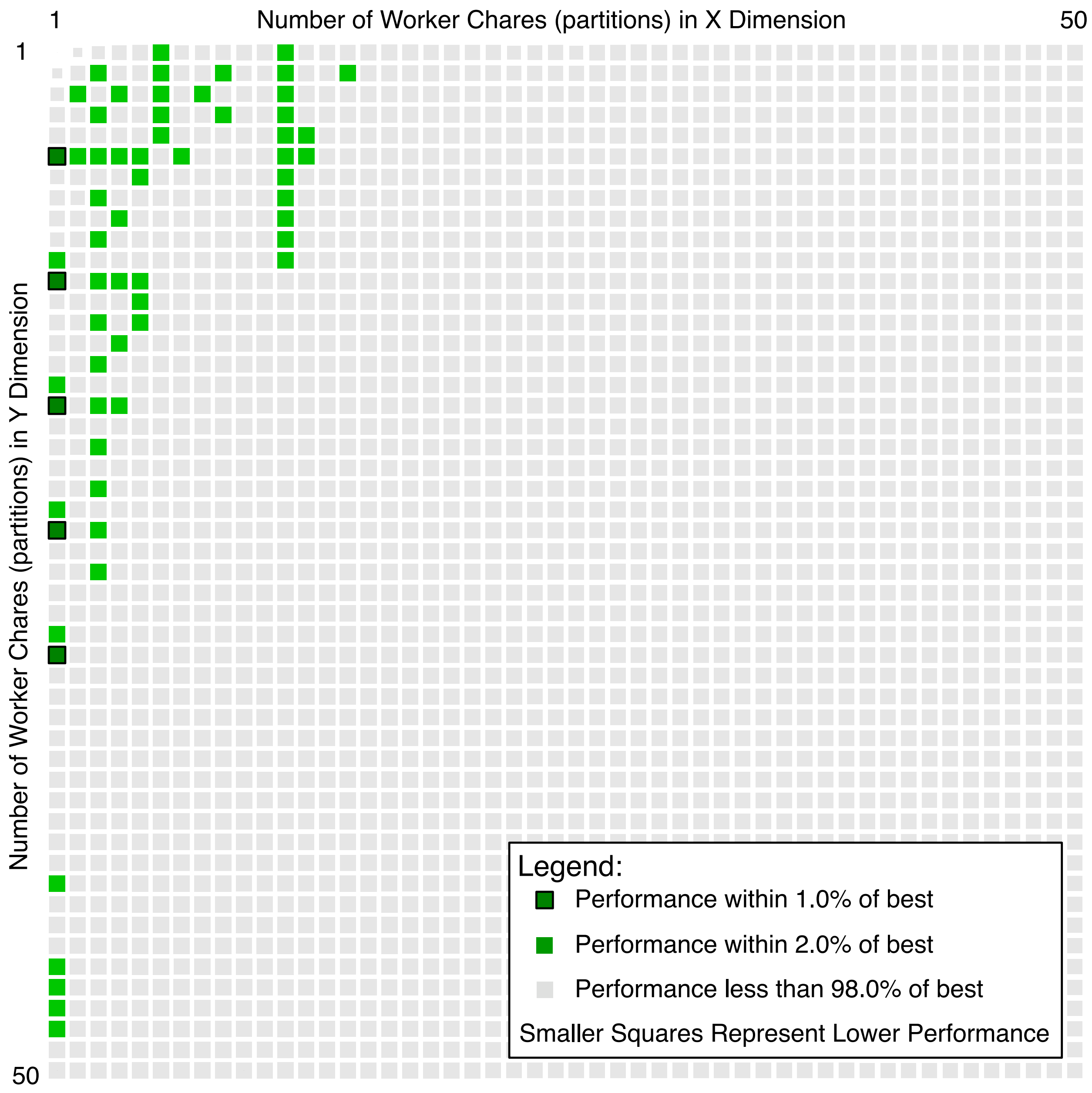

Figure 5: Performance for all possible parameter space configurations for the Wave2D program on a 4000×4000 data grid on 6 processors of an 8-core Intel Linux workstation. In this case, only a few of the tested configurations are close to optimal. The best performing decompositions of the 2-D grid data is to slice it into $6, 12, 18, 24$, or 30 slices. For this case, blocked decompositions are not as good as the sliced decompositions.

4 Conclusion

This paper presents a new performance tuning framework, and examines the use of the preliminary version of the framework to optimize the performance of two programs. The approach for the tuning framework is to provide feedback from an adaptive runtime system to the program so that the program can modify its behavior. This feedback and adaptation will provide new opportunities for optimizing the whole system's performance, not just that of the application code by itself.

Results are provided for exhaustive searches of the control point spaces for the two test cases. These exhaustive searches were performed on two platforms; the structure of optimal performance regions for this program is simple for the Pipelined Stream Filtering program, but is complicated for the Wave2D program.

Some of the future work we envisage includes studies of multiple algorithms to identify potential control points, programmer-provided characterization of control point ranges, and smart adaptive runtime strategies that examine the current performance data to select a subset of knobs to turn, and when possible, the direction to turn each, so as to heuristically improve the performance.

References

[1] Sanjeev Krishnan. *Automating Runtime Optimizations For Parallel Object-Oriented Programming*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.

[2] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, 1993.

[3] L. V. Kale, Sameer Kumar, and Krishnan Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS'03*, Nice, France, April 2003.

[4] Ahmad Faraj, Xin Yuan, and David Lowenthal. Star-mpi: self tuned adaptive routines for mpi collective operations. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 199–208, New York, NY, USA, 2006. ACM.

[5] Matthew J. Koop, Terry Jones, and Dhabaleswar K. Panda. Mvapich-aptus: Scalable high-performance multi-transport mpi over infiniband. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '08)*, April 2008.