An Adaptive Algorithm Selection Framework*

Hao Yu
IBM T. J. Watson Research Ctr
Yorktown Heights, NY 10598
yuh@us.ibm.com

Dongmin Zhang
Parasol Lab, Dept of CS
Texas A&M University
dzhang@cs.tamu.edu

Lawrence Rauchwerger
Parasol Lab, Dept of CS
Texas A&M University
rwerger@cs.tamu.edu

Abstract

Irregular and dynamic memory reference patterns can cause performance variations for low level algorithms in general and for parallel algorithms in particular. We present an adaptive algorithm selection framework which can collect and interpret the inputs of a particular instance of a parallel algorithm and select the best performing one from a an existing library. In this paper present the dynamic selection of parallel reduction algorithms. First we introduce a set of high-level parameters that can characterize different parallel reduction algorithms. Then we describe an off-line, systematic process to generate predictive models which can be used for run-time algorithm selection. Our experiments show that our framework: (a) selects the most appropriate algorithms in 85% of the cases studied, (b) overall delievers 98% of the optimal performance, (c) adaptively selects the best algorithms for dynamic phases of a running program (resulting in performance improvements otherwise not possible), and (d) adapts to the underlying machine architecture (tested on IBM Regatta and HP V-Class systems).

1. Introduction

Improving performance on current parallel processors is a very complex task which, if done "by hand" by programmers, becomes increasingly difficult and error prone. Programmers have obtained increasingly more help from parallelizing (restructuring) compilers. Such compilers address the need of detecting and exploiting parallelism in sequential programs written in conventional languages as well as parallel languages (e.g., HPF). They also optimize data layout and perform other transformations to reduce and hide memory latency, the other crucial optimization in modern large scale parallel systems. The success in the "conventional" use of compilers to automatically optimize code is limited to the cases when performance is independent of the input data of the applications. When codes are irregular (memory references are irregular) and/or dynamic (change during the execution of the same program instance) it is very likely that important performance affecting program characteristics are input and environment dependent. Many important (frequently used and time consuming) algorithms used in such programs are indeed input dependent. We have previously shown [25] that, for example, parallel reduction algorithms are quite sensitive to their input memory reference pattern and system architecture. In [1] we have shown that parallel sorting algorithms are sensitive to architecture, data type, size, etc. One of the most powerful optimizations compilers can employ is to substitute entire algorithms instead of trying to perform low level optimizations on sequences of code. In [1] and [25] we have shown that performance improves significantly if we dynamically select the best algorithm for each program instance.

In this paper, we present a general framework to automatically and adaptively select, at run-time, the best performing, functionally equivalent algorithm for each of their instantiations. The adaptive framework can select, at run-time, the best parallel algorithm from an existing library. The selection process is based on an off-line automatically generated prediction model and algorithm input data collected and analyzed dynamically. For this paper we have concentrated our attention on the automatic selection of reduction algorithms. For brevity we will not show in this paper the dynamic selection of parallel sorting algorithms.

Reductions (aka updates) are important because they are at the core of a very large number of algorithms and applications – both scientific and otherwise – and there is a large body of literature dealing with their parallelization. More formally, a reduction variable is a variable

* Research was performed at Texas A&M and supported in part by NSF CAREER Award CCR-9734471, NSF Grant ACI-9872126, NSF Grant EIA-0103742, NSF Grant ACI-0326350, NSF Grant ACI-0113971, DOE ASCI ASAP Level 2 Grant B347886

whose value is used in one associative and possibly commutative operation of the form $x = x \otimes exp$, where $\otimes$ is the operator and x does not occur in exp or anywhere else in the loop. With the exception of some simple methods using unordered critical sections (locks), reduction parallelization is performed through a simple form of algorithm substitution. For example, a sequential summation is a reduction which can be replaced by a parallel prefix, or recursive doubling, computation [12]. In the case of irregular, sparse programs, we define *irregular reductions* as reductions performed on data referenced in an irregular pattern, usually through an `index` array. For *irregular reductions*, the access pattern traversed by the sequential reduction in a loop is often sensitive to the input data or computation. Therefore, as we have shown in [25], not all parallel reduction algorithms or implementations are equally suited as substitutes for the original sequential code. Each access pattern has its own characteristics and will best be parallelized with an appropriately tailored algorithm.

In [25] we have presented a small library of parallel reduction algorithms and shown that the best performance can be obtained only if we dynamically select the most appropriate one for the instantiated input set (reference pattern). Also in [25] we have presented a taxonomy of reduction reference patterns and sketched a decision tree (*derived manually*) based scheme.

We continue this work and introduce a systematic and automatic process to generate predictive models that match the parallel reduction algorithms to execution instances of reduction loops. After *manually* establishing a small set of parameters that can characterize irregular memory references and setting up a library of parallel reduction algorithms [25], we measure their relative performance for a number of memory reference parameters in a factorial experiment. This is achieved by running a synthetic loop which can generate reduction references with the memory reference patterns selected by our factorial experiment. The end result of this off-line process is a mapping between various points in the memory reference pattern space and the best available reduction algorithm. At run-time, the memory reference characteristics of the actual reduction loop are extracted and matched through a regression to the corresponding best algorithm (using our previously extracted map).

This paper's main contribution is a framework for a systematic process through which input sensitive predictive models can be built off-line and used dynamically to select from a particular list of functionally equivalent algorithms, parallel reductions being just on important example. The same approach could also be used for various other compiler transformations that cannot be easily analytically modeled. Our experiments on an IBM Regatta and HP V-Class show that the framework: (a) selects the best performing algorithms in 85% of the cases studied, (b) overall delievers 98% of the optimal performance, (c) achieves better performance by adaptively selecting the best algorithm for the dynamic phase of a running program, (d) adapts to machine architecture.

2. Framework Overview

In this section we give an overview of our general framework for adaptive and automatic low level algorithm selection, the details of which are presented in the remainder of this paper as applied to the optimization of parallel reduction algorithm selection.

Fig. 1 gives an overview of our adaptive framework. We distinguish two phases: (a) a setup phase and (b) a dynamic selection (optimization) phase.

The **setup phase** occurs once for each computer system and thus, implicitly tailors our process to a particular architecture. We then establish the input domain and the output domain (possible optimizations) of the algorithm selection code.

In the particular case presented in this paper, i.e., parallel reductions, the input domain is the universe of all possible and realistic memory reference patterns — because, because they crucially impact the obtained performance [25]. Architecture type is also important, but is used implicitly. Since it would be impractical to study the entire universe of memory access patterns, we define a small set of parameters that can sufficiently characterize them. The domain of possible optimizations is composed, in our case, of the different parallel reduction algorithms collected in a library. Here, we need to note that the characteristic parameters for selecting algorithms for different optimizations can be different. To avoid extensive setup process, high level parameters (usually need to be identified manually) are preferred. For instance, the characteristic parameters for selecting parallel sorting algorithms [1] are different from what we choose for reduction parallelization.

We then explore our input domain and find a mapping to the output domain. In our case we establish a mapping between different points in the input parameter space (memory reference patterns) and relative performance rankings of the available algorithms. This task is accomplished off-line by running a factorial experiment. We generate a number of parameter sets that have the potential cover our input domain. For each of these data points, we measure the relative performance of our algorithms on the particular architecture, and rank them accordingly. We should mention here that we have also tried other methods of exploring the data space.

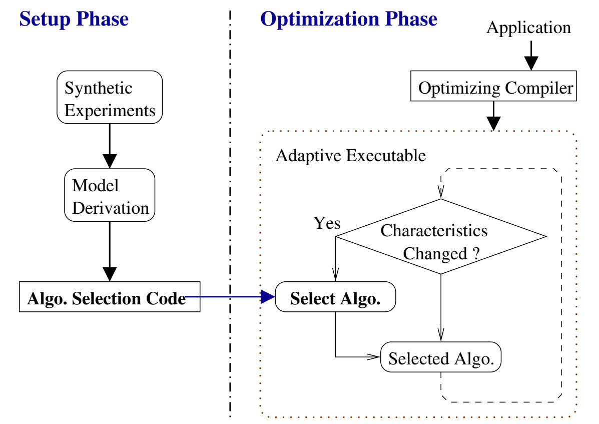


Figure 1. Adaptive Algorithm Selection

In [24, 1], we have used a machine learning algorithmto explore the input space that defines performance.

The **dynamic selection phase** occurs during actual program execution. Through instrumentation (or otherwise) we extract the set of relevant parameters that characterizes the actual input. Then we used the information obtained during the setup phase to find its corresponding output. Specifically, we use statistical regression to eventually select the appropriate best performing parallel reduction algorithm. In [1] and [24], in turn, we used statically generated decision trees which are dynamically traversed to select the best algorithms.

In this paper we specialize our adaptive framework to parallel reduction algorithm selection. In the following, we first present our library of reduction algorithms and a set of parameters that can characterize their irregular memory reference patterns. Then we describe the factorial experiment that explores our input space. For each parameter set, we execute a synthetic parameterized loop that generates a memory reference pattern on which we evaluate and rank the different algorithms in our library. A regression method is used to compute prediction models for each parallel reduction algorithm based on the data from the factorial experiment. At runtime, we compute values for the characterization parameters of the reduction operation in question and use them in our pre-computed models to select the best algorithmic option. We then evaluate our framework experimentally for static and dynamic reference patterns.

3. Reduction Algorithm Library

Our parallel reduction algorithm library contains several methods suitable for a range of reference patterns. For brevity, we provide here only a high level description of the methods. See [25] for additional details.

Reductions are associative recurrences and they can be parallelized in several ways. Our library currently contains two types of methods: *direct update methods*, which update shared reduction variables during loop execution, and *private accumulation and global update methods*, which accumulate in private storage during loop execution and update shared variables with each processor's contribution afterwards.

Direct update methods include the classical *recursive doubling* [12], *unordered critical sections* [26] and *local write* [7]; our library only includes local write because the others are not as competitive. **Local Write** (LOCALWR) is originally described in [7], LOCALWR first collects the reference pattern in an *inspector loop* [19] and partitions the iteration space based on the "owner–computes" rule. Memory locations referenced on multiple processors have their iterations replicated on those processors as well. To execute the reduction(s), each processor goes through its local copies of the iterations containing the reduction(s) and operating on the processor's local data.

We have implemented three private accumulation methods in our library. **Replicated Buffer** (REPBUF) [12, 14] simply executes the reduction loop as a `doall`, accumulating partial results in private reduction variables, and later accumulates results across processors. **Replicated Buffer With Links** (REPLINK) [25] avoids traversing the unused (but allocated) elements during REPBUF's cross-processor reduction phase by keeping a linked list to record the processors that access each shared element. **Selective Privatization** (SELPRIV) [25] only privatizes array elements that have cross-processor contention. By excluding unused private elements, SELPRIV maintains a dense private space where most elements are used. Since the private space does not align to the shared data array, the reduction's index array is modified to redirect accesses to the selectively privatized elements.

Issues	REPBUF	REPLINK	SELPRIV	LOCALWR
Good when contention is	High	Low	Low	Low
Locality	Poor	Poor	Good	Good
Need schedule reuse [19]	No	Yes	Yes	Yes
Extra Work	No	No	No	Yes
Extra Space	`NxP`	`NxP`	`NxP+M`	`MxP`

`M`: # iterations; `N`: data array size; `P`: # processors.

Table 1. Comparison of reduction algo.

In Table 1, we present a qualitative comparison of the algorithms described above. Here, the contention of a reduction is the average number of iterations (# pro-

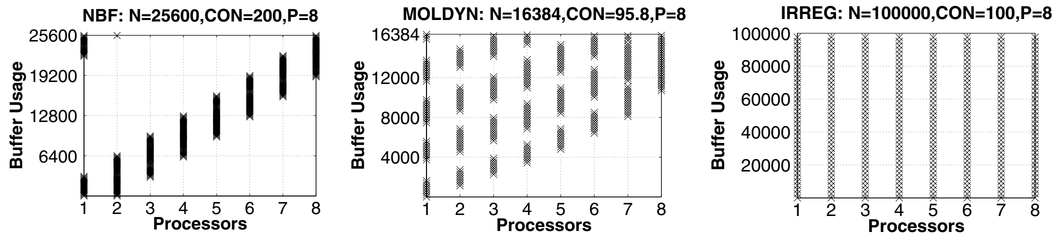


Figure 2. Memory access patterns of RepBuf

cessors when running in parallel) referencing the same element. When contention is low, many unused replicated elements in RepBuf are accumulated across processors, while other algorithms only pass useful data. SelPriv works on a compacted data space and therefore potentially has good spatial locality. In LocalWr, each processor works on a specific portion of the data array and has thus potentially good locality. With respect to overhead, RepLink, SelPriv and LocalWr all have auxiliary data structures that depend on the access pattern, and which must be updated when it changes. Their overhead can be reduced by employing the *schedule reuse* [19] technique.

4. High-Level Parameters

In this section, we describe the parameters we have chosen to characterize reduction operations. Ideally, they should require little overhead to measure and they should enable us to select the best parallel reduction algorithm from our library for each reduction instance in the program. After defining the identified parameters below, we present a summary of the decoupled effects of the parameters on the performance of the parallel reduction algorithms to illustrate the effectiveness of the chosen parameters. Here, we enumerate the parameters in no specific order.

N is the *number data elements* involved in the reduction (often the size of the reduction array). It strongly influences the loop's working set size, and thus potentially performance. In some applications, several reduction arrays have exactly same access pattern; here **N** includes the data elements for all arrays.

CON, the *Connectivity* of a loop, is the ratio between the number of iterations of the loop and **N**. This parameter is equivalent to the parameter defined by Han and Tseng in [7]; there, the underlying data structures (corresponding to the irregular reductions) represent graphs $G = (V, E)$ and the authors defined **Connectivity** as $|V|/|E|$. Generally, the higher the connectivity, the higher the ratio of computation to communication, i.e., if the connectivity is high, a small number of elements will be referenced by many iterations.

MOB, the *Mobility* per iteration of a loop, is the number of distinct subscripts of reductions in an iteration. For the LocalWr algorithm, the effect of high iteration Mobility (actually lack of mobility) is a high degree of iteration replication. MOB is a parameter that can be measured at compile time.

OTH, represents the *Other (non-reduction) work* in an iteration. If **OTH** is high, a penalty will be paid for replicating iterations. To measure **OTH**, we instrumented a parallel loop transformed for the RepBuf algorithm using light-weight timers ($\sim$ 100 clock cycles).

SP, the *Sparsity*, is the ratio of the total number of referenced private elements and total allocated space on all processors using the RepBuf algorithm (p**N**). Intuitively, **SP** indicates if RepBuf is efficient.

CLUS, the *Number of Clusters*, reflects spatial locality and measures if the used private elements in the RepBuf are scattered or clustered on each processor. Fig. 2 shows three memory access patterns which can be classified as *clustered*, *partially-clustered*, and *scattered*, respectively. Currently, **SP** and **CLUS** are measured by instrumenting parallel reduction loops using the RepBuf algorithm and the overhead is proportional to the number of used private elements. **CLUS** measures the average number of clusters of the used private elements on each processor.

We have investigated the decoupled effects of the parameters on the performance of the parallel reduction algorithms. Although the decoupling is not realistic, it is useful for discovering qualitative trends. The trends summarized in Table 2 are based on a comprehensive set of experiments on multiple architectures that are reported in [24]. The trends for RepLink are similar to RepBuf and are not listed here.

The effect of **N** is straight–forward, comparing to the sequential reduction loop, SelPriv and LocalWr have much smaller data sets on each processor and therefore their speedups increase with **N**. **CON** is inversely correlated with inter-processor communication. Hence, larger **CON** values will indicate better scaling RepBuf

parameters	RepBuf	SelPriv	LocalWr
N	↗	↑	↑
CON	↑	↗	–
MOB	↑	↘	↓
OTH	↑	↑	↗
SP	↗	↓	↘
CLUS	↗	↑	–

↑: positive effect; ↓: negative effect; ↗: little positive effect; ↘: little negative effect; –: no effect.

Table 2. Decoupled effects

and SelPriv, which have two reduction loops, one accumulating in private space and one accumulating cross-processor shared data. Large **MOB** values (many references to `index` arrays) may imply poor performance for SelPriv, because accesses to the reduction array must access both the original and the modified index arrays, and for LocalWr, because large **MOB** values often result in high iteration replication. Large values of **OTH** often indicate good performance for RepBuf and SelPriv because the first private accumulation loop has a larger iteration body; since LocalWr replicates "other work" it will not benefit as much. Low **SP** is good for SelPriv and also for LocalWr, since it often correlates with low contention and hence low iteration replication. For large **CLUS**, since SelPriv compacts the sparsely used data space, it achieves better speedups than LocalWr and RepBuf, which work on original (non-compacted) data or in private spaces conformable to the original data.

5. Adaptive Reduction Selection

In this section we will elaborate on the **setup** and on the **dynamic selection** phases of our adaptive scheme. The **setup** is executed only once, during machine installation, and generates a map between points in the universe of all inputs (memory reference patterns characterized with the previously defined parameters) and their corresponding best suited reduction algorithms. The **dynamic selection** phase is executed every time a targeted reduction is encountered. It uses the map built during the setup phase, a parameter collection mechanism to characterize the memory references and an interpolation function (the actual algorithm selector) to find the best suited algorithm in the library.

5.1. Setup Phase

We now outline the design of the initial map between a set of synthetically generated parameter values and the corresponding performance ranking of the various paral-

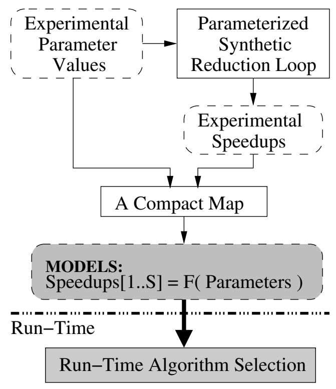

Experimental Parameter Values
Parameterized Synthetic Reduction Loop
Experimental Speedups
A Compact Map
MODELS:
Speedups[1..S] = F(Parameters)
Run−Time
Run−Time Algorithm Selection

Figure 3. Setup phase

lelization algorithms available in our library. The overall *setup phase*, is illustrated in Fig. 3.

The domain of values that can be taken by the input parameters is explored by setting up a factorial experiment [10]. Specifically, we choose several values (typical of realistic reduction loops) for each parameter and generate a set of experiments from all combinations of the chosen values of the different parameters. The chosen parameter values for our reduction experiment are shown in Table 3.

To measure the performance of different reduction patterns, we have created a synthetic reduction loop. The structure of the loop is shown in Fig 4, with C-like pseudo-code. The non–reduction work and reductions have been grouped in two loop nests. The dynamic pattern depends strictly on the `index` array, which is generated automatically to correspond to the parameters **SP** and **CLUS**. In addition, **OTH** is dynamically measured. The contents of the `index[*]` array are generated via a randomized process which satisfies the requirement specified by the parameters.

The performance ranking of the parallel reduction algorithms in our library is accomplished by simply executing them all for each parameter combination, i.e., synthetically generated access pattern, and measuring their actual speedups, as shown in Fig. 3. The end result is a compact map from parameter values to performance (speedups) of candidate parallel algorithms.

From this map we can now create the prediction code, the *model* that can then be used by an application at run-

```
FOR j = 1 to N*CON
  FOR i = 1 to OTH  /* non-reductions */
    memory read & scalar computation;
  FOR i = 1 to MOB      /* reductions */
    data[ index[j][i] ] += expr;
```

Figure 4. Synthetic reduction loop

time. We applied two methods, *general linear regression* and *decision tree learning*. However, due to space constraints, in this paper we will describe only the modeling process using *general linear regression*.

Parameters	Values				
N (array size)	16K	64K	256K	1M	4M
Connectivity	0.2	2	16	128	
Mobility	2	8			
Other Work	1	4			
Sparsity	0.02	0.2	0.45	0.75	0.99
# clusters	1	4	20		

Table 3. Param. for factorial experiment

The generated models are polynomial functions from the parameter values to the speedups of the parallel algorithms. We follow a standard *term selection* process that automatically selects polynomial terms from a specified pool [16]. We have specified a maximal model as $F = (\lg N + \lg C + MOB + OTH + \lg S + \lg L + 1)^3$. and a minimal model as $F = \lg N + \lg C + MOB + OTH + \lg S + \lg L + 1$. Here, C is **CON**, S is **SP**, and L is **CLUS**. Then from the minimal model, terms are randomly selected from the 84 terms of the maximal model.

The samples are first separated into training data and testing data. When adding a term into the model, the training data are used to fit (least square fit) the coefficients and the fitted model is evaluated using the testing data. If the test error is higher than the test error of the model before including the new term, the term will be dropped. Though this sequential term selection process will not give us the optimal model, it is fast and its automatically generated models have produced good results when used to predict relative performance of the parallel algorithms on real reduction loops. Here, the order of the model, 3, is chosen mainly due to practical reasons such as generating less experimental samples with less synthetic experimentation time and avoiding term explosion. For parameter **MOB**, we have only chosen 2 values for the experiment, we have excluded the terms containing a non-linear **MOB** factor from the maximal model. For **OTH** and **CLUS**, though we specified few values, the values used in the map are measured and computed from the generated `index` array.

The final polynomial models contain about 30 terms. The corresponding C library routines are generated automatically to evaluate the polynomial $F()$ for each algorithm at run-time.

5.2. Dynamic Selection Phase

During the dynamic selection phase, to avoid executing the parameter collection and algorithm selection steps for every time a reduction loop is invoked, we use a form of memoization, *decision reuse*, which detects if the inputs to our selector function have changed. When a new instance of a reduction is encountered and the input parameters have not changed from the previous execution instance, we directly reuse the previously selected algorithm, thus saving run-time selection overhead. This is accomplished with standard compiler technology, i.e., the compiler instruments two version loops for each parallel algorithm (illustrated in Fig. 5).

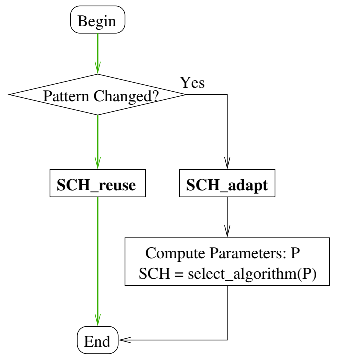


Figure 5. Reduction selection at run-time

In this phase, the pre-generated model evaluation routines are called to estimate speedups of all the algorithms, rank them, and select the best one. The evaluation of the polynomial models is fast because each of the final models contains only about 30 terms.

5.3. Selection Reuse for Dynamic Programs

In the previous section we presented a systematic process to generate predictive models that could recommend the best irregular reduction parallelization algorithm by collecting a set of static and dynamic parameters. We mentioned that if the memory characteristics parameters do not change we can reuse our decision and thus reduce run-time overhead. This may be of value if the compiler can automatically prove statically that the reference pattern does not change. If, however this is not possible, which is often the case for irregular dynamic codes, we have to perform a selection for each reduction loop instance. In this section we will show how to reduce this overhead by evaluating a trade-off between the run-time overhead of selection and the benefit of finding a better algorithm.

To better describe the run–time behavior of a dynamic program, we introduce the notion of **dynamic phase**. For a loop containing irregular reductions, a **phase** is composed of all the contiguous execution instances for which the pattern does not change. For instance, assume the access pattern of the irregular reduc-

Program	Lang.	Lines	Source	Loop	Coverage	# inp
Irreg – FE based CFD kernel	F77	223	[7]	apply elements' forces (do 100)	$\sim$ 90%	4
Nbf – MD kernel	F77	116	[7, 21]	non-bounded force calc. (do 50)	$\sim$ 90%	4
Moldyn – synthetic MD program	C	688	[7, 4, 15]	non-bounded force calc.	$\sim$ 70%	4
Charmm – MD kernel	F77	277	[8]	non-bounded force calc.	$\sim$ 80%	3
Spark98 – FE simulation	C	1513	SPEC'2000	smvp loop – Symmetric MV Product	$\sim$ 70%	2
Spice – circuit simulation	F77	18912	SPEC'92	bjt loop – traverse BJT devices & update mesh	11–45%	4
Fma3d – FE method for solids	F90	60122	SPEC'2000	Scatter_Element_Nodal_Forces_Platq loop	$\sim$ 30%	1

Table 4. Scientific Applications and Reduction Loops

tions changes at instance t and the next change of the pattern is at instance t', the loop instances in $[t, t' - 1]$ form a **dynamic phase**. We can therefore group the execution instances of irregular reduction loops into **dynamic phases**. We define the *re-usability* of a phase as the number of dynamic instances of the reduction loop in that phase. It intuitively gives the number of times a loop can be considered invariant and thus does not need a new adaptive algorithm selection. That is, the overhead of selecting a better algorithm at run-time can be amortized over *re-usability* instantiations.

We then include *re-usability* into our previously developed models, predict the phase-wise performance and thus obtain a better overall performance. This model will be employed at run-time to decide when it is worth changing the algorithm. The phase–wise speedup of a parallel algorithm can be formulated as $\frac{(R-1)T_{par}+(1+O)T_{par}}{T_{seq}}$. The best algorithm is the algorithm that has the smallest value of $\frac{R+O}{Speedup}$. In above formulas, R is the *re-usability*, O is the ratio of the set-up phase overhead (in time) and the parallel execution time of one instance of the parallel loop applying the considered algorithm, and $Speedup$ is the speedup (relative to sequential execution) of the considered algorithm excluding the set-up phase overhead. T_{par} and T_{seq} are the parallel and sequential execution times of the loop. Using the same off-line experimental process described previously, we have generated models to compute O, the setup overhead ratio. Together with the predicted speedup (excluding the setup overhead), we were able to evaluate $\frac{R+O}{Speedup}$ at run–time and select the best algorithm for a dynamic execution phase.

The *re-usability* **R** can be either computed by the compiler if it can statically detect the phase changes, i.e., the number of iterations of the surrounding loop for which the address pattern remains constant (a well known technique named 'schedule reuse' [19]) or estimate on-the-fly with a simple statistical method (e.g., running average value of **R**). In our experiments we have used the schedule reuse technique and the simple statistical method. As a general rule, simulations do not change phase very fast so re-usability is high.

6. Evaluation of Algorithm Selection

In this section we evaluate our automatically generated performance models. We compare the actual performance of the algorithm selected by our automatically generated predictive models with the other algorithms.

6.1. Experimental Setup

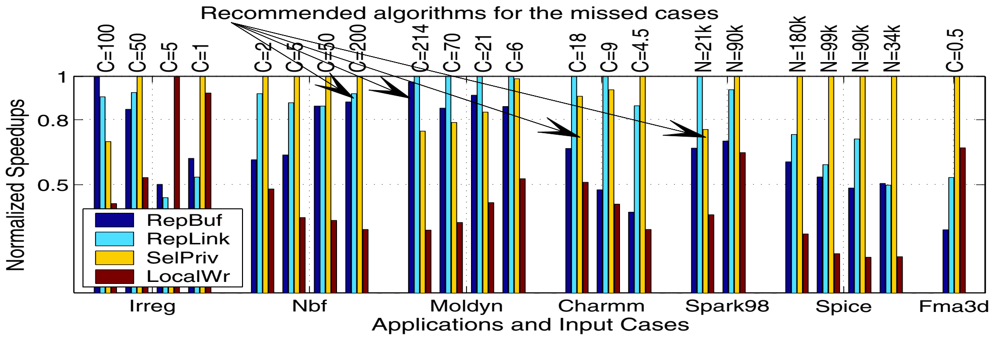
We selected seven programs from a variety of scientific domains (see Table 4). All Fortran codes were automatically instrumented to collect information about the access pattern using the Polaris compiler [2]; the C programs were done manually. For most programs, we chose or specified multiple inputs to explore input-dependent behavior. While specifying the inputs, *we have tried, where possible, to use data sets that exercise the entire memory hierarchy of our parallel machine.*

We have studied two parallel systems: an UMA HP V-Class with 16 processorsand a NUMA IBM Regatta system with 32 processors.The machine configurations are briefly described in Table 5.

	HP V2200	IBM Regatta
CPU Type	PA-8200	POWER 4
CPU Clock	200 MHz	1300 MHz
Data Cache	2 MB	32KB/1.48MB/32MB
Memory	4 GB	64 GB
# CPUs	16	32 / 4 MCMs
Topology	crossbar	buses / ring
OS	HP-UX 11.0	AIX 5
Compiler	HP f90, c89	xlf_r, xlc_r

Table 5. Experimental parallel systems

We used an 8-processor subsystem of the HP and a 16-processor subsystem of the IBM, which was sufficient for us to exercise the architectural characteristics of these two systems. We ran 22 application/input combinations on the HP and due to resource limitations only 21 application/input cases on the IBM system (no results were obtained for `FMA3D`).



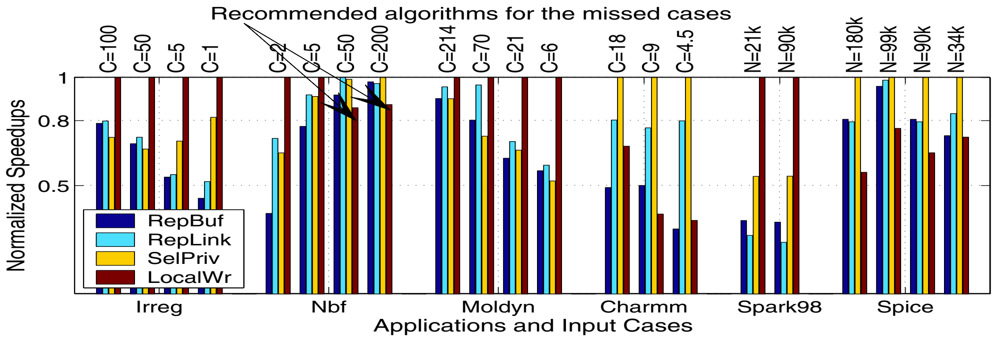


Figure 6. Relative speedups of parallel reduction algorithms on HP and IBM systems

6.2. Results

Fig. 6 presents the results obtained on both systems. Each group of bars shows the relative performance (normalized to the best speedup obtained for that group) of the four parallel reduction algorithms for one program/input case. In most cases, the algorithm rankings resulting from the automatically generated regression model were consistent with the actual rankings. Overall, our regression model correctly identified the best algorithm in 18/22 cases on the HP and 19/21 cases on the IBM. As the arrows in the graphs show, for all the mis-predicted cases there was little performance difference between the best and recommended algorithms.

To give a quantitative measure of the performance improvement obtained using the algorithms recommended by our models, we compute the *relative speedup* which we define as the ratio between the speedup of the algorithm chosen by an "alternative selection method" and the algorithm recommended by "our model". We have compared the effectiveness of our predictive models against four "alternative selection methods". **The Best** is a "perfect predictive model", or an "oracle", that always selects the best algorithm for a given loop–input case. **RepBuf** always applies REPBUF, which is the simplest algorithm and it is specified as default in `OpenMP` [17]. **Random** randomly selects a parallel algorithm. The speedup obtained is the average speedup of all the candidate parallel algorithms for that case. **Default** always applies the "default" parallel algorithm for a given platform. Based on our experimental results, we chose SELPRIV and LOCALWR as the default algorithms for the HP and IBM systems, respectively.

Fig. 7 gives the average *relative speedups* (across all the loop-input cases). In the graph, the smaller the *speedup ratio* value, the better the relative effectiveness of our predictive models. The graph shows that our automatically generated predictive models work almost as well as the "perfect predictive models", obtaining more than 98% of the best possible performance. Comparing to other "non-perfect" selection methods, our predictive models improve the performance significantly.

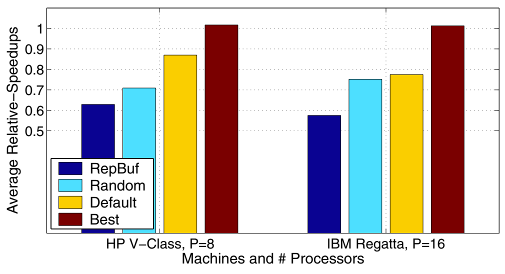
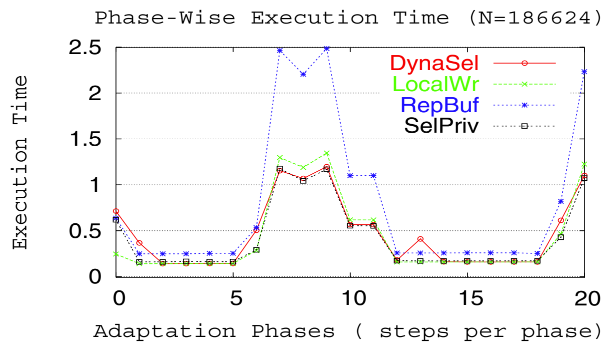


Figure 7. Average relative-speedups



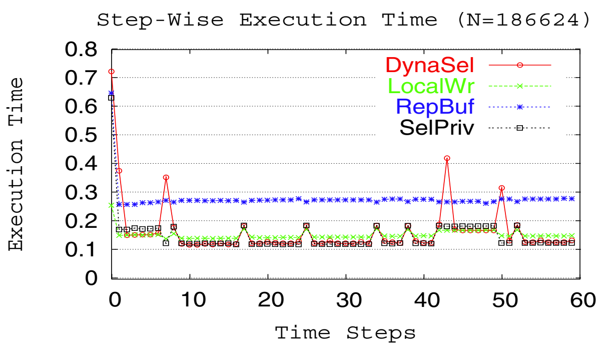


Figure 8. Effects of dynamically selecting algorithms of MOLDYN (input #4)

As mentioned previously, the overhead of measuring **OTH** is negligible by using a light-weight timer in few iterations; the overhead of computing **SP** and **CLUS** is proportional to the size of the data. Across all cases, the average overheads on the HP and IBM systems are, respectively, 11.95% and 11.65%, which can be largely amortized since the overhead only incurs in loop instances where the reduction pattern change.

Finally, we note that knowledge of the dynamically collected parameters (**N**, **CON**, **OTH**, **SP** and **CLUS**) is very important for our models. From Fig. 6, especially for the bars corresponding to the results for `Irreg` and *Charmm* on the HP system, the best schemes for different inputs change and our technique predicts the best schemes correctly by collecting and utilizing the dynamically collected parameters.

To test the robustness and generality of our predictive models, we have applied them to two regular reduction loops: loop `loops_do400` in `su2cor` from the SPEC'92 suite, and loop `actfor_do500` in `bdna` from the PERFECT suite. For regular reduction loops, the size of the reduction data (usually a vector) is relative small and every element of the vector is accessed in every loop iteration. Either selectively replicating the vector data or partitioning the vector will not help performance (reducing cross–processor communications). The experimental results indicate that REPBUF is always the best algorithm and that our predictive models have given the correct recommendations. Hence, our *adaptive algorithm selection* technique is generally applicable to all reductions.

7. Experiments on Dynamic Programs

In this section, we present experimental results showing that our adaptive algorithm selection technique can select the best parallelization scheme dynamically and improve the overall performance of adaptive programs. Here, DYNASEL represents applying algorithm selection dynamically for every computation-phase.

7.1. Molecular Dynamics (MOLDYN)

MOLDYN is a synthetic benchmark, conducting non-bonded force calculations in a molecular dynamics simulation. It has been widely used for the purpose of evaluating systematic optimization transformations [7, 4, 15]. For the program used in our experimentation, we replaced the step in MOLDYN that building a reusable neighbor list with a *link-cell* algorithm [18]. With the modification, the execution time of MOLDYN is dominated by the force computation, which is the irregular reduction we would like to optimize.

ID	#molecules	cut-off radius	avg. CON	#steps	#phases
1	23328	4.0	157	60	23
2	186624	2.5	37.6	60	23
3	100800	2.0	21.8	60	21
4	186624	1.5	6.6	60	21
5	186624	1.2	5.4	60	21

Table 6. Inputs of MOLDYN

For MOLDYN, since we have not observed much performance change across dynamic phases for parallel algorithms, we artificially set the *re-usability* (the number of steps) for each dynamic phase so that the phase-wise best algorithms can change due to the different setup overheads of the algorithms. This way, we can examine both the effectiveness of the prediction models and the efficiency of selecting and switching algorithms. We experiment with 5 different inputs on the 8-processor subsystem of our HP V-class machine. The input specifications are given in Table 6. Note that different phases may have different numbers of time steps.

Due to space limitations, we show the step-wise and phase-wise execution time Fig. 8 for one input. We note that since the number of steps of the phases may change, therefore in the phase-wise plots, we plotted the accumulated time of each phase instead of the average time. Again, the results show that DYNASEL out-performs applying any one algorithm for most cases.

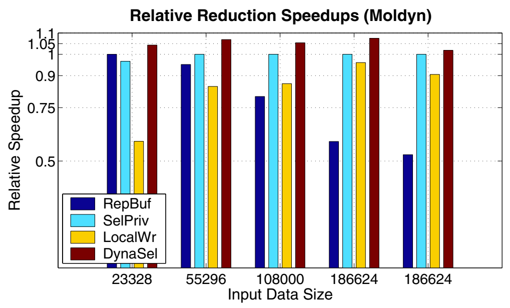


Figure 9. Results for MOLDYN

Fig. 9 gives the relative speedups of the reduction loop across all steps for different parallel reduction algorithms. The speedups are normalized to the best results obtained by applying one algorithm to clarify the improvement. The conclusions here are that using our adaptive technique to select and apply the best parallel reduction algorithm for every dynamic phase has little overhead and that it out-performs any other single algorithm and improves overall performance up to 8%;

7.2. PP2D in FEATFLOW

We have applied our adaptive technique to the PP2D code from the FEATFLOW package [20]. FEATFLOW is a general purpose, F77 library of subroutines for solving incompressible Navier-Stokes equations in 2D and 3D. More specifically, PP2D solves nonlinear coupled equations using multi-grid solvers. It has about 17,000 lines of code, excluding supporting library routines.

After profiling for our test input set, we have selected a loop with irregular reductions from subroutine GUPWD (about 11% of whole program execution time). The loop updates a sparse matrix with old velocity values associated with grid nodes. The memory access pattern of the irregular reduction is fully determined by the indirection data structures defining the sparse matrix. The program uses a multi-grid method to solve linear systems in each time step. The multi-level grids are predefined (specified in the input data file) and the sparse matrix structures associated with the different grids are defined in an initialization step, before the time-step loop. For our input data the program used 4 grids, with grid levels between 2 and 5 (in general, the code could use as many as 9 grid sizes). The studied reduction loop uses the 4 sparse-matrix data structures (the four grids) in an interleaved manner. The matrices themselves, i.e., the reduction access pattern, do not change, only their interleaved invocation does. We then apply our adaptive algorithm selection technique for every invocation of the reduction loop. By using the well known *schedule reuse* [19] technique we can record our selection for each of the four reduction patterns corresponding to each of the four grids used by the code. The selection decision is then reused for subsequent invocations of the loop. In this manner we reduce the overhead associated with dynamic selection to only 4 instances (the 4 grids do not change during program).

Detailed information about the used input, which is distributed together with the source code for benchmarking purposes, is shown in Table 7.

Level	#unknowns	#nodes	#elements	#instances
2	12930	1890	920	86
3	52620	7460	3680	86
4	206280	29640	14720	86
5	824720	118160	58880	166

Table 7. Grid parameters of PP2D input

Fig. 10 gives the speedups of the GUPWD loop obtained, for each grid level, using three reduction techniques, as well as the dynamic selection scheme. All bars in the graph are normalized to the optimal scheme.

The group 'Total' shows the normalized performance of the loop, across all invocations, for the case when the schemes would be selected only at the beginning of the program as well as for the dynamic selection case when a decision is made for every loop instance but the decision is reused.

On the HP system, although our DYNASEL has selected the best algorithm (SELPRIV for all the grid levels), it did not improve the overall performance of the loop because the SELPRIV scheme performs best in all cases. Noteworthy is the fact that the overhead of the dynamic scheme has not hurt performance.

On the IBM system, the results show that the performance of DYNASEL align to the actual best algorithm for each grid level. More importantly though, DYNASEL obtains the overall best performance because the choice of the optimal algorithm varies across grid levels, i.e., across the instances of the reduction loop. If we would select only once the best scheme then performance could degrade when the grid level changed. The use of the *decision reuse* technique, keeps the effect of dynamic selection overhead to a minimum.

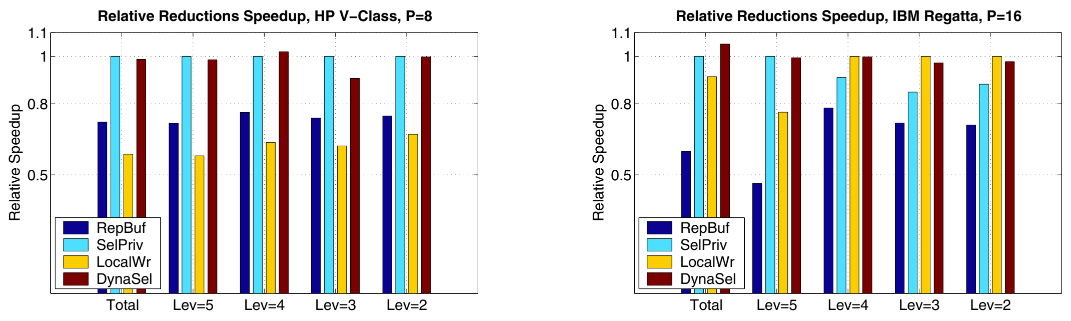


Figure 10. Results for PP2D

This experiment shows that we can apply our dynamic algorithm selection method even to dynamic programs in an always profitable manner. It shows that even if performance improvement in our particular case was modest (upto 8%), our framework can take advantage of instance specific opportunity for optimization.

8. Related Work

There does not appear to be a great body of effort in the area of adaptive selection of high level algorithms. Brewer[3] is probably the most extensive previous work aimed at making a framework for this decision process. In this approach, performance models are consulted to determine the expected running time of possible implementation, with the minimum running algorithm then chosen for use. These models are linear systems given, along with some code annotations, by the end user.

Li, Garzaran, and Padua[13] present an approach for choosing between several sequential sorting algorithms based on data size, data entropy, and an installation benchmarking phase that correctly selects the best algorithm for the given situation. However, no attempt is made to generalize the technique into a general approach and no discussion of the more difficult parallel case is given. Many previous work aim at tuning specific algorithm parameters. Examples are Spiral[23] and FFTW[6] for FFT signal processing and Atlas[22] for matrix multiplication. These approaches, though quite effective, are very narrow in scope and do not constitute a general framework for generic algorithm selection.

Another somewhat relevant approach is that of *dynamic feedback* [5] which selects code variants based on on-line profiling. There are many recent efforts to develop dynamic and adaptive compilation systems, however we feel they are out of scope for this work.

Both *data affiliated loop* [14] method and *Local Write* [7] follow "owner computes" rule. Disadvantages of these techniques are their potential to heavily replicate unnecessary computation and cause load imbalance, unlike data replication based methods.

Zoppetti and Agrawal [27] proposed a parallel reduction algorithm for multi-threaded architectures that can overlap computation and communication. They also implemented an incremental inspector that overlaps computation and communication and update the computation schedule efficiently. The potential drawback of this technique is that it may introduce unnecessary communication, e.g., pipelining data sections to irrelevant computation threads.

Adaptive Data Repository (ADR) infrastructure [11] was developed to perform range queries with user-defined aggregation operations on multi-dimensional datasets, which are generalized reductions. In the *ADR* infrastructure, three strategies are used: fully replicated accumulation, sparsely replicated accumulation, distributed accumulation, which are analogous to REPBUF, SELPRIV and LOCALWR discussed in this paper. Their experiments have shown that none of the strategies worked the best for various query patterns and a predictive model was desired.

9. Conclusion

In this paper, we presented an *Adaptive Algorithm Selection Framework* that can automatically adapt to the input data, environment and machine and select the best performing algorithm. We have applied our framework to the adaptive selection of parallel reduction algorithms. We have identified a few high-level, architecture independent parameters characterizing a program's static structure, dynamic data access patterns and candidate transformations. We applied an off-line synthetic experimental process to automatically generate predictive models which are used to dynamically select the most appropriate optimization transformation among several functionally equivalent candidates.

Through experimental results, we have shown that our technique can select the most appropriate parallel reduction algorithms at run–time with very low overhead. When this technique is applied to dynamic programs selecting new algorithms for each of their dynamic phases performance is improved even further.

The importance of this work is that the presented adaptive optimization technique can model programs with irregular and dynamic behavior and customize solutions to each program instance. It is a general framework that can adapt any number of optimizations to the program's needs.

References

[1] P. An and et al. STAPL: An adaptive, generic parallel c++ library. In *Proc. Wkshp. Lang. Compilers for Parallel Computing*, pp. 193–208, 2001.

[2] W. Blume and et al. Advanced program restructuring for high-performance computers with Polaris. *IEEE Computer*, 29(12):78–82, 1996.

[3] E. A. Brewer. High-level optimization via automated statistical modeling. In *Proc. ACM Symp. Principles and Practice of Parallel Prog.*, pp. 80–91, 1995.

[4] C. Ding and K. Kennedy. Improving cache performance of dynamic applications with computation and data layout transformations. In *Proc. ACM Conf. Prog. Lang. Design and Implementation*, pp. 229–241, 1999.

[5] P. C. Diniz and M. C. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proc. ACM Conf. Prog. Lang. Design and Implementation*, pp. 71–84, 1997.

[6] M. Frigo. A fast fourier transform compiler. In *Proc. ACM Conf. Prog. Lang. Design and Implementation*, pp. 169–180, 1999.

[7] H. Han and C.-W. Tseng. Improving compiler and runtime support for adaptive irregular codes. In *Proc. IEEE Int'l Conf. Parallel Architectures and Compilation Techniques*, pp. 393–400, 1998.

[8] Y.-S. Hwang and et al. Runtime and language support for compiling adaptive irregular programs on distributed-memory machines. *Software - Practice and Experience*, 25(6):597–621, 1995.

[9] R. Iyer, N. M. Amato, L. Rauchwerger, and L. Bhuyan. Comparing the memory system performance of the HP V-Class and SGI Origin 2000 multiprocessors using microbenchmarks and scientific applications. In *Proc. ACM Int'l Conf. Supercomputing*, pp. 339–347, 1999.

[10] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., Hoboken, NJ, 1991.

[11] T. Kurc and et al. Querying very large multi-dimensional datasets in adr. In *Proc. Supercomputing '99*, 1999.

[12] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, San Francisco, CA, 1992.

[13] X. Li, M. J. Garzaran, and D. A. Padua. A dynamically tuned sorting library. In *Proc. 2004 Int'l Symp. Code Generation and Optimization*, pp. 111–122, 2004.

[14] Y. Lin and D. A. Padua. On the automatic parallelization of sprase and irregular fortran programs. In *Proc. Int'l Wkshp. Lang., Compilers, and Run-time Systems for Scalable Computers*, pp. 41–56, 1998.

[15] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *Int'l J. Parallel Prog.*, 29(3):217–247, 2001.

[16] A. Miller. *Subset Selection in Regression (Second Edition).* Chapman & Hall/CRC, Boca Raton, FL, 2002.

[17] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface, Version 2.0*, 2000.

[18] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J. Comp. Phys.*, 117:1–19, 1995.

[19] J. H. Saltz, R. Mirchandaney, and K. Crowley. Runtime parallelization and scheduling of loops. *IEEE Trans. Computers*, 40(5):603–612, May 1991.

[20] S. Turek and C. Becker. *FEATFLOW: Finite Element Software for The Incompressible Navier-Strokes Equations, User Manual, Release 1.1.* University of Heidelberg, Institute for Applied Mathmatics, Germany, 1998.

[21] R. von Hanxleden. Handling irregular problems with Fortran D – a preliminary report. In *Proc. Wkshp. Compilers for Parallel Computers*, pp. 353–364, 1993.

[22] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–25, 2001.

[23] J. Xiong, J. Johnson, R. Johnson, and D. A. Padua. SPL: A language and compiler for dsp algorithms. In *Proc. ACM Conf. Prog. Lang. Design and Implementation*, pp. 298–308, 2001.

[24] H. Yu, F. Dang, and L. Rauchwerger. Parallel reduction: An application of adaptive algorithm selection. In *Proc. Wkshp. Lang. Compilers for Parallel Computing*, pp. 171–185, 2002.

[25] H. Yu and L. Rauchwerger. Adaptive reduction parallelization techniques. In *Proc. ACM Int'l Conf. Supercomputing*, pp. 66–77, 2000.

[26] H. P. Zima. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, NY, 1991.

[27] G. M. Zoppetti, G. Agrawal, and R. Kumar. Compiler and runtime support for irregular reductions on a multithreaded architecture. In *CDROM Proc. Int'l Parallel and Distributed Processing Symp.*, 2002.