# A Cost Model for Communication on a Symmetric MultiProcessor *

Nancy M. Amato[†] Andrea Pietracaprina[‡] Geppino Pucci[§]

Lucia K. Dale[†] Jack Perdue[†]

Technical Report 98-004
Department of Computer Science
Texas A&M Univerity
January 26, 1998

## Abstract

In this paper we conduct an in-depth study of the communication costs of programs when run on a typical Symmetric MultiProcessor, the SGI Power Challenge, characterized by powerful off-the-shelf microprocessors communicating through a shared memory via a shared-bus interconnect. Our study is based on an extensive set of experiments designed to assess the relative impact of a number of parameters on the cost of shared memory accesses. We provide evidence that interaction with the memory hierarchy affects communication in such a substantial way that none of the models previously considered in the literature can guarantee a reasonable level of accuracy since they do not take this interaction into account. We then determine two prediction functions that are very accurate predictors of best and worst performance with respect to the memory hierarchy. These functions provide a prediction interval that can be employed to obtain lower and upper bounds on the actual communication cost of an application, and to evaluate the degree of locality of the memory access patterns involved.

*This research was supported in part by NATO CRG 961243 "Bulk Synchronous Computational Geometry," and by the National Center for Supercomputing Applications under CCR970010N (utilizing the SGI Power ChallengeArray at NCSA, University of Illinois at Urbana-Champaign). The work at Texas A&M was also supported by the NSF by CAREER award CCR-9624315 and grant IRI-9619850.

[†]Department of Computer Science, Texas A&M University, College Station, TX, USA.

[‡]Dipartimento di Matematica Pura e Applicata, Università di Padova, Italy.

[§]Dipartimento di Elettronica e Informatica, Università di Padova, Italy.

# 1 Introduction

The widespread use of parallel computers has been hampered by the difficulty of exploiting their massive computational potential to an extent that warrants their large cost. This is in stark contrast to the vast body of ingenious parallel algorithms developed over the last two decades. Indeed, it has often been noted that theoretically efficient algorithms exhibit poor performance when implemented on real machines. This is due not only to the intrinsic limitations of asymptotic analysis but also to the inadequacies of the cost model employed to assess their complexity. These inadequacies have received considerable attention over the last decade, often within the context of the more general quest for a *bridging model* of parallel computation, i.e., one that balances conflicting requirements such as simplicity, accuracy and generality.

Although much progress has been made, the development of adequate tools for predicting actual algorithm performance on real machines remains one of the most challenging problems in parallel processing. We believe that further progress towards this goal requires a tighter coupling of the cost model to the architecture than has been previously employed. In this paper we investigate the issue of predictivity for a specific architecture belonging to the family of symmetric multiprocessors. In particular, we study the impact of the memory hierarchy on performance and determine under which conditions simple cost functions can guarantee accurate performance predictions.

**Bulk-Synchronous Models** One of the first and most popular attempts to define a bridging model was made by Valiant [Val90], who introduced the *Bulk-Synchronous Parallel (BSP)* model. The model abstracts a parallel machine as a set of processors with local memory, connected through a communication medium of limited bandwidth. The computation is organized as a sequence of *supersteps* separated by barrier synchronizations, where processors operate asynchronously in a superstep. The cost of a superstep is modeled as a linear function of the largest amount of communication performed by a single processor, and the function coefficients relate to a few architectural parameters (bandwidth and synchronization overhead). The large body of work this model has generated has proved its suitability for the development of portable software (see e.g., [GHM+96]). However, as has been often observed, the simple BSP cost model offers only a coarse level of predictivity, since it disregards architectural features of real machines which may have a dramatic impact on performance [BGMZ97, JW96].

Similar in spirit to BSP, but based on different programming paradigms, are the CG Model [DFRC93], which focuses on algorithmic rather than architectural issues, and the LogP model [CKP+96], a fully asynchronous model whose performance predictions tend to be more accurate than those provided by BSP but are still subject to the same limitations [BHP+96, BGMZ97].

The BSP opened the way to a rich line of research which resulted in the definition of a number of variants that maintain the basic bulk-synchronous approach to parallel programming but try to enhance the predictive quality of the associated cost model. Among these, we mention the BSP* [BDM95] and the E-BSP [JW96], which extend the BSP cost model to account for *block transfer* and *unbalanced communication*, respectively. Both these models target distributed-memory architectures, where communication is realized via message passing. Bulk-synchronous models specifically tailored for shared-memory systems are the $(d, x)$-BSP [BGMZ97] and the QSM [GMR97], whose cost functions account for some aspects of memory contention, namely, the maximum number of concurrent accesses to the same memory location (QSM), and the maximum number of concurrent accesses to the same memory bank ($(d, x)$-BSP). It must be remarked, however, that none of these models accounts for the influence of the memory hierarchy on the running time, except for the distinction between local and global memory accesses.

**Importance of Memory Hierarchy for SMPs** Although an application's interaction with the memory hierarchy is an important factor on any system, it is particularly crucial for shared-memory machines such as the *Symmetric Multiprocessors (SMPs)*, a generation of machines characterized by off-the-shelf powerful microprocessors interacting through a (perhaps distributed) shared-memory via a shared communication

network (e.g., a bus). In such a system, the cost of an access may vary dramatically: from 1-2 cycles if the data is in first-level cache (L1), to tens of cycles for second-level (L2) cache, to hundreds of cycles if the data must be brought from main memory. The cost may be even greater in the presence of invalidations or false-sharing (different processors accessing distinct data that reside in the same cache line). Thus, it is clear that on such a system the *locality of reference* (spatial or temporal) of an application can greatly impact its running time.

Another architectural feature of some SMPs which is usually disregarded by cost models, thus making performance predictions inaccurate, is that read and write accesses do not necessarily have the same cost. In particular, read accesses may be *blocking* (computation halts until the data is obtained) whereas write accesses may not (they may be buffered allowing computation to proceed).

**Our Results** The objective of this work is to provide a meaningful cost model for communication on the SGI PowerChallenge (PC) system, which is a distinguished representative of the class of modern SMPs. In an effort to analyze the effects of the memory hierarchy on communication, we study two extreme families of access patterns that make the best-case and worst-case use of the memory hierarchy by maximizing and minimizing, respectively, the locality of reference. The results of our investigation are summarized in the following points:

- *The influence of the memory hierarchy is crucial.* Access patterns from the two families that are identical with respect to the number of reads/writes performed by each individual processor, have execution costs that differ by up to two orders of magnitude on the SGI PC. This implies that accurate performance predictions cannot be obtained with cost models that disregard caching effects (as is the case for all previous shared-memory bulk-synchronous models).

- *Accuracy can be attained by controlling caching effects.* Nevertheless, within each family of access patterns we show that communication costs can be predicted with a high-level of accuracy through a BSP-like linear cost function of the 'right' parameters. For each family, we consider several cost functions to assess the relative importance of a number of parameters. We provide evidence that the most consistently accurate functions are those which account separately for reads and writes, and include a measure of the overall communication volume. (A similar conclusion was drawn in [JW96] for parallel systems based on message-passing.)

- *Prediction Interval.* The cost functions synthesized for the two families of access patterns provide a prediction interval that can be used to: (i) lower and upper bound an application's actual communication time, and (ii) evaluate an application to assess the degree of locality of reference of its communication patterns. Thus, the prediction interval may be employed as a profiling tool to gain valuable insights into the application's interaction with the memory hierarchy, which may in turn lead to better algorithm design [SRG94].

Our conclusions are based on an extensive set of experiments designed to identify the quantities that affect the cost of communication and to assess the relative impact of these quantities. In order to study communication in isolation, we consider a bulk-synchronous programming model, similar in spirit to the one adopted in [GMR97], in which a program is organized as a sequence of *supersteps*, where in a superstep, local computation and reads/writes to the shared memory are made in distinct phases separated by barriers.

We measure actual costs of three different suites of access patterns for each family, and use one suite to interpolate the cost functions and the others to assess the predictive quality of the functions. This allows us to select two best functions (whose predictions are always within 5-7% of actual values), one for each family, and to use these functions to form the prediction interval. We also run three different sorting algorithms and verify that measured costs always fall within the prediction interval, which provides evidence that the two families do indeed make extreme use of the memory hierarchy. In addition, we illustrate the use of the prediction interval as an evaluation tool by analyzing the communication patterns embodied in some of the supersteps of the sorting algorithms.

**Comparison with related work.** We want to stress that the main purpose of this paper is to study the cost of communication on a specific machine, the SGI Power Challenge, and to establish under which conditions the performance of an algorithm can be accurately predicted on such a machine based on quantities that are easily obtainable from the algorithm. (We believe, however, that our findings apply more generally to the class of SMPs). In this respect, our objective differs from those that motivated other works concerned with modeling shared-memory systems [GMR97, BGMZ97].

The QSM model proposed in [GMR97] attempts to strike a balance between descriptivity and generality, with the intent to enhance the effectiveness of algorithm design while maintaining a high level view of a machine. The model is architecture independent, hence its cost function aims at highlighting factors that influence algorithm performance rather than providing accurate predictions. Also, the difference between QSM and BSP predictions, mainly due to the new contention parameter introduced by QSM, can be appreciated only when the number of available processors, which is an upper bound to this parameter, is large. Therefore, QSM predictions for machines with very few processors (e.g., many SMPs) tend to be no more accurate than those provided by BSP.

On the other hand, the $(d, x)$-BSP [BGMZ97] aims at a higher level of descriptivity for a certain class of shared-memory machines, characterized by numerous but slow memory banks and by the provision of latency-hiding features to compensate for bank delays, as is the case, for example, of vector multiprocessors. In such machines, memory bank contention has a considerable impact on performance, especially for irregular access patterns, and therefore it is included as a parameter in the cost model. However, bank contention may not fully be controllable at the algorithmic level, since it depends on aspects, such as the memory map, that are not necessarily specific to the architecture but rather to the run-time support (e.g., consider the case when memory locations are hashed among the banks). Furthermore, as acknowledged by the authors, the $(d, x)$-BSP does not capture memory hierarchy effects, which crucially affect performance on SMPs.

## 2 The Architecture

The SGI Power Challenge (SGI PC) [Sil95] is a shared-memory multiprocessor architecture. The NCSA system used in our study [Ncs97] is based on the MIPS superscalar RISC R10000 chip, which is a 64-bit processor that uses the MIPS IV instruction set. The cache system consists of a 32 KB on-chip instruction cache, a 32 KB on-chip level-1 (L1) data cache, and a 2 MB off-chip unified (instructions and data) level-2 (L2) cache. The length of the L1 and L2 cache lines are 32 and 128 bytes (or 8 and 32 integers), respectively. The clock speed and processor cycle time on the SGI PC are 195 MHz and 5.1 ns, respectively.

The NCSA system used has 16 processors which communicate using a 4 GB distributed shared-memory via a fast shared-bus interconnect. The bus has a bandwidth of 1.2 GB per second with a 256-bit wide data bus and a separate 40-bit wide address bus that can access up to one terabyte (TB) of physical memory. The bus provides high-bandwidth, low-latency, cache-coherent communication between processors, memory, and I/O. The operating system is IRIX 6.2, which is based on Unix System V.

## 3 Programming Model

In broad terms, a *Single Program Multiple Data* (SPMD) computation on an SMP is characterized by a single program which is executed in parallel by all processors. The program may be thought of as a standard sequential program (e.g., C code) where the communication among the processors is realized by accessing the shared memory. We distinguish between *local* and *global accesses*, and between *local* and *global variables*. A local variable exists in multiple copies which are made local to the processors, and each processor operates on its own copy without interfering with other processors. Conversely, a global variable is unique and all processors access the only available copy for that variable. As a consequence, accesses to

local variables do not involve processor interaction, while communication is realized only through global accesses.

As mentioned in the introduction, we will conduct our study referring to a bulk-synchronous programming style [Val90, GMR97]. More specifically, we regard a program as made of a sequence of *supersteps*, where each superstep consists of three consecutive *phases*: *copy-in*, *local computation* and *copy-out*. In the copy-in phase, each processor transfers the contents of the global variables needed for its computation into local ones. In the local computation phase, the processor works exclusively on local data, and in the copy-out phase, the processor updates the contents of the global memory to reflect the results of this computation. At the end of each phase, a barrier is enforced so that local computation is never mixed with global accesses. By doing so, we are able to precisely account for the fraction of time that a program devotes to communication.

On the SGI Power Challenge we implemented the above programming model by writing parallel programs using standard sequential C code where the SPMD programming paradigm is realized through the SGI native `m_fork` and `m_sync` primitives. An `m_fork` takes a function as a parameter and executes it in SPMD mode on all processors. Any variables declared inside the function are regarded to be local, while variables declared outside the function are considered global. (In a set of experiments that, for brevity, are not reported in this abstract, we compared the costs of local vs global accesses, and observed the former to be negligible with respect to the latter.) The `m_sync` primitive implements a barrier and relies on a hardware synchronization mechanism whose cost is quite low, averaging less than 200 $\mu s$ for 12 processors. In our programs, the supersteps correspond to `m_fork` calls, and the three phases within a superstep are separated by `m_sync`'s. No `m_sync` is used between supersteps since the `m_fork` already enforces a synchronization upon termination.

## 4 Cost Models

Our objective is to study the communication cost of a superstep when run on an SGI PC, providing, when possible, simple metrics to estimate this cost[1]. We identify the following factors as those that mostly affect the running time of the copy-in and copy-out phases within a superstep.

1. The maximum number of global reads/writes performed by any processor;
2. The total number of global reads/writes performed by all processors;
3. The benefits and penalties introduced by the memory hierarchy, as a function of the access pattern.

While the first two factors can be easily quantified as numerical parameters, the effects of the memory hierarchy are more complex and hard to capture in a quantitative fashion. In fact, the cost of a sequence of global reads/writes is highly dependent on the *locality of reference*, [ACS87], which in turn relates to the level of the hierarchy being accessed, the contiguity of consecutive accesses, and the *write conflicts* among the processors, which activate the invalidation mechanism of the cache coherence protocol.

**Good and Bad Access Pattern Families.** As indicated by practice, and as will be further confirmed by our experiments, the interaction with the memory hierarchy can dramatically affect the cost of a global access pattern, even when all other parameters are constant. To study this issue within the framework provided by our programming model, we consider two extreme families of access patterns, referred to as *Good* and *Bad*, which access the shared memory in such a way as to maximize benefits and penalties, respectively, caused by the interaction with the memory hierarchy. Let $p$ denote the number of processors. For any given superstep, let $hr_i$ (resp., $hw_i$) denote the number of global reads (resp., writes) performed by Processor $i$ in

[1]We remark that it is outside the scope of the paper to provide an accurate prediction model for the time required by local computation, and refer the reader to [HP96] for an extensive treatment of this topic.

that superstep, with $1 \leq i \leq p$. Let $t_{\max}$ denote an upper bound on the maximum number of reads/writes performed by a processor ($2 \cdot 10^6$ in our case), and let $t_{\text{line}}$ denote the number of integers that fit in a cache line (32 for L2 cache on our SGI PC). In our experiments, reads and writes are always performed on a shared array of integers.

We say that the superstep is executed in *good mode* (i.e., the access pattern it realizes is in the Good family) if Processor $i$ reads (resp., writes) the (consecutive) integers stored in the array locations with indices $(i-1)t_{\max} + k$, for $0 \leq k < hr_i$ (resp., $0 \leq k < hw_i$). Additionally, prior to the execution of the superstep all processors' caches are primed to guarantee that each processor performs the largest possible number of accesses at the highest levels of the hierarchy (i.e., L1, then L2, then memory). In this way, locality of reference is maximized while no cache invalidations occur.

Conversely, the superstep is executed in *bad mode* (i.e., the corresponding access pattern is in the Bad family) if Processor $i$ reads (resp., writes) the array locations with indices $i + kt_{\text{line}}$, for $0 \leq k < hr_i$ (resp., $0 \leq k < hw_i$). In this way, locality of reference is minimized, since a processor never accesses more than one integer on the same cache line, and the number of invalidations involved in the pattern is maximized as a consequence of the high contention on the cache lines.

Thus, a superstep can be thought of as embodying a particular access pattern in its copy-in and copy-out phases. In the next subsections, we provide experimental evidence that within the Good and Bad families, the communication cost of a superstep can be accurately predicted.

### 4.1 Experiments

We consider four processor configurations of the SGI PC, namely $p = 2, 4, 8$ and 12, and perform a number of experiments for every configuration, that exercise a large spectrum of access patterns, i.e., combinations of read and write counts. Each experiment consists of the execution of a superstep in either good or bad mode. We ran three distinct suites of experiments, called *Suite 1*, *Suite 2* and *Suite 3*, respectively. Let $\mathcal{H}_0 = \{5000 * i \;:\; 1 \leq i \leq 10\}$, $\mathcal{H}_1 = \{50000 * i \;:\; 1 \leq i \leq 10\}$ and $\mathcal{H}_2 = \{550000 + 150000 * i \;:\; 0 \leq i \leq 9\}$, and let $\mathcal{H} = \mathcal{H}_0 \bigcup \mathcal{H}_1 \bigcup \mathcal{H}_2$.

**Suite 1** For every $h \in \mathcal{H}$ and $1 \leq x < p$, the following three supersteps are run in good and in bad mode:

`like-gather`$(x, h)$**:** $hr_i = h$ if $1 \leq i \leq x$, and 0 otherwise; $hw_i = hx/p$ for every $1 \leq i \leq p$.

`like-scatter`$(x, h)$**:** $hr_i = hx/p$ for every $1 \leq i \leq p$; $hw_i = h$ if $1 \leq i \leq x$, and 0 otherwise.

`vary`$(x, h)$**:** $hr_i = hw_i = h$ for every $1 \leq i \leq x$.

The above terminology reflects the fact that `like-gather` privileges reads, `like-scatter` privileges writes, and `vary` balances reads and writes while changing, as $x$ grows, the overall number of accesses done by all processors. Note that when $x = p$ the three types of supersteps coincide. In this case, we execute only one of them.

**Suite 2** For every $h \in \mathcal{H}$ and $1 \leq x < p$, three supersteps are run both in good and bad mode, assigning the reads and writes to be done by each processor at random but making sure that the number of reads/writes performed by each processor is less than or equal to (at least one holds with equality) the maximum number in `like-gather`$(x, h)$, `like-scatter`$(x, h)$ and `vary`$(x, h)$, respectively. In this fashion, we preserve the variety of combinations of maximum read and write counts exercised in Suite 1, but randomize the selection of the specific access pattern.

**Suite 3** For every $h \in \mathcal{H}$ and $1 \leq x < p$, three supersteps are run both in good and bad mode, assigning the reads and writes to be done by each processor at random but making sure that the total combined number of reads and writes performed by all processor is as in `like-gather`$(x, h)$, `like-scatter`$(x, h)$ and `vary`$(x, h)$, respectively. This suite is similar but somewhat orthogonal to Suite 2, in the sense that here we fix the total rather than maximum number of reads and writes.

### 4.2 Cost Functions

We investigate the predictive quality of a number of cost functions in order to determine the best trade-off between simplicity (i.e., the minimum number of parameters involved) and accuracy. The form of these functions is inspired by the BSP cost model and its variants, in that they all consist of a constant term $L$ and a linear combination of a set of parameters, related to the number of reads and writes performed by the processors. Each function has been obtained through least-square fitting using the data gathered in one suite, and its quality has been assessed by measuring the average and maximum relative errors between predicted and actual running times of the supersteps in the other two suites.

#### 4.2.1 Good Family

Let $hr$ (resp., $hw$) denote the maximum number of reads (resp., writes) done by any processor in the copy-in (resp., copy-out) phase of a superstep, and let $M$ denote the total number of reads and writes done in the superstep. Also define $h = \max\{hr, hw\}$. For the Good family of supersteps and for each processor configuration, we fit the following functions:

`H`$(h)$: $L + g_h \cdot h$;

`HM`$(h, M)$: $L + g_h \cdot h + g_M \cdot M$;

`HrHw`$(hr, hw)$: $L + g_{hr} \cdot hr + g_{hw} \cdot hw$;

`HrHwM-c`$(hrc, hrm, hwc, hwm, M)$: $L + g_{hrc} \cdot hrc + g_{hrm} \cdot hrm + g_{hwc} \cdot hwc + g_{hwm} \cdot hwm + g_M \cdot M$.

In the `HrHwM-c` function, we split the contribution of $hr$ and $hw$ into two terms, namely $hrc$ and $hrm$ for $hr$, and $hwc$ and $hwm$ for $hw$. The terms $hrc$ and $hwc$ respectively account for reads and writes done on global variables that reside in L2 cache at the beginning of the superstep, while the terms $hrm$ and $hwm$ account for the remaining global reads and writes. On our SGI PC, where the L2 cache contains a maximum of $2^{19}$ integers, we considered the first $2^{19}$ reads/writes as contributing to $hrc/hwc$, and the remaining ones as contributing to $hrm/hwm$ (recall that caches are primed prior to the superstep).

The functions have been obtained from Suite 1 data, and validated on Suite 2 and Suite 3 data. Moreover, in order to improve predictivity, we subdivide the Good supersteps into two sets $R0$ and $R1$ based on the value of $h$, and fit distinct functions for each of the two sets. Specifically, $R0$ contains all supersteps with $h \leq 2^{19}$ (which all have $hrm = hwm = 0$) and $R1$ contains those with $h > 2^{19}$. The coefficients of all functions for $p = 8$ are shown in Table 1, while, for brevity, we only report the coefficients of function `HrHwM-c` (which was found to be the most accurate function) for the other processor configurations (see Table 2). (Note that the cost functions express running times in $\mu s$.) The average and maximum relative errors between actual and predicted times for the supersteps of Suites 2 and 3, are reported in Tables 5 and 6 for all functions and all values of $p$.

#### 4.2.2 Bad Family

The functions we consider to predict the costs of supersteps belonging to the Bad family are:

`H`$(h)$: $L + g_h \cdot h$;

`HM`$(h, M)$: $L + g_h \cdot h + g_M \cdot M$;

`HrHw`$(hr, hw)$: $L + g_{hr} \cdot hr + g_{hw} \cdot hw$;

`HrHwM`$(hr, hw, M)$: $L + g_{hr} \cdot hr + g_{hw} \cdot hw + g_M \cdot M$.

As opposed to the Good family, it was unnecessary to distinguish between cache accesses and memory accesses in function `HrHwM`, since the Bad family comprises extreme access patterns that do not take any significant advantage of the cache. The functions have been obtained by fitting Suite 2 data, and validated on Suites 1 and 3. As for the Good family, we report the coefficients of all functions only for $p = 8$ (Table 3), while we report the coefficients of `HrHwM` (which was the most accurate function) for the other processor configurations (see Table 4). The average and maximum relative errors between actual and predicted times of the supersteps of Suite 1 and 3, are reported in Tables 7 and 8 for all functions and all values of $p$.

### 4.3 Analysis of Results

A number of observations concerning the cost functions and their estimated predictivity are in order. In general, we note that not all functions exhibit the same predictive quality, although average errors never exceed 30%. In particular, it is easy to see that function `H`, which is analogous to the BSP cost model, exhibits the poorest predictivity (e.g., see the graphs in Figure 2). This is due to the fact that reads and writes have, in general, different costs. Thus, unifying their contributions into a single parameter $h$ results in an inevitable loss of accuracy. On the other hand, function `HrHwM-c` for the Good family, and function `HrHwM` for the Bad family, which separate the contributions of reads and writes and introduce the total number of accesses as an additional parameter, provide very accurate predictions that are always within 7% of the actual values. (Graphs showing predicted against measured times for a sample of patterns from both Good and Bad families are given in Figure 1.)

This validates our hypothesis that when cache effects are fixed, as is the case for the two families of access patterns considered, the cost of communication can be accurately predicted by a linear function of the maximum number of accesses performed by any processor and the total number of accesses performed by all processors. Moreover, the accuracy of the predictions, hence the linearity of the communication costs, improves as $h$ increases. Indeed, for the Good family, the prediction errors of function `HrHwM-c` for the set R1 are better than those for R0. Similarly, for the Bad family, a number of tests we made, which are not reported in this abstract, show that the prediction errors of function `HrHwM` tend to decrease as $h$ increases. Specific considerations regarding the cost functions for each family are made below.

**Good family** A number of conclusions can be drawn by looking at the coefficients of the `HrHwM-c` function. When accesses are done in cache, reads are about twice as expensive as writes (compare $g_{hrc}$ vs $g_{hwc}$). This is explained by the fact that on the SGI PC reads in cache are blocking while writes are not. Moreover, in the Good family cache invalidations, which would increase the cost of the writes, are avoided. In contrast, for accesses in main memory the difference between reads and writes fades away.

The coefficients of the functions appear rather insensitive to the number of processors, since a Good access pattern involves very little interference. One exception is the slight decrease of $g_{hrc}$ and $g_{hwc}$ for $R1$ as $p$ increases, which is compensated by an increase of $g_M$. Note however that these three coefficients have little weight compared to $g_{hrm}$ and $g_{hwm}$. Also, for any fixed processor configuration, the functions for $R0$ and $R1$ look very similar, except for the $L$ term, which plays a second order role and whose value is very sensitive to noise in the data. This suggests that the distinction of the supersteps in the two sets $R0$ and $R1$ may not be necessary. In fact, we can show that by fitting the `HrHwM-c` function on all supersteps, average prediction errors become only slightly worse and are always below 10% (more details will be provided in the full version).

In summary, by looking at the validations of the different functions, we can clearly conclude that the BSP-like function `H` is consistently worse and `HrHwM-c` is consistently better than the others. As for the other two functions `HM` and `HrHw`, we observe that `HrHw` is significantly better than `HM` except for $R1$ and $p \geq 8$, where the latter becomes more accurate. Again, this can be explained by the fact that when accesses are entirely done in cache (case $R0$) there is little interference of requests on the bus, and thus the parameter $M$ plays almost no role, while it is important to distinguish between reads and writes, which have different costs. On the other hand, for larger $h$ (case $R1$) more memory accesses are done, thus $M$ becomes more

important (since all memory accesses compete for the same resource), while the distinction between reads and writes fades away.

**Bad family** Let us now concentrate on the functions fit for the Bad family, with particular reference to `HrHwM`, whose predictive quality is definitively superior to that of the other functions. The first observation is that the impact of global reads and writes on the running time of a superstep is now reversed with respect to the Good family, since writes are weighted about twice as much as reads (compare $g_{hr}$ vs $g_{hw}$). This reflects the fact that a large number of write misses and invalidations take place during the execution of a superstep in bad mode. For the same reason, the coefficient $g_M$ of the $M$ parameter is much larger than the one in the `HrHwM-c` function of the Good family, since parameter $M$ is an indicator of the high level of contention among the processors. Indeed, as $p$ increases, the $g_{hr}$ and $g_{hw}$ coefficients decrease while $g_M$ increases, since the effects of contention are better captured by $M$, which is related to the overall number of accesses, rather than by $hr$ and $hw$, which describe the activity of a single processor.

As in the case of the Good family, the BSP-like function `H` is consistently worse than all other functions, while `HrHwM` is consistently better. Indeed, the gap in predictive quality between $H$ and the other functions is more pronounced than before, since function `H` has no means of accounting for the different costs of reads and writes *and* for the greater impact of $M$ on the running time. Function `HrHw` is slightly more accurate than function `HM` for $p \leq 4$ processors, while the latter becomes more accurate than the former for $p \geq 8$. This phenomenon substantiates our previous observation that, as $p$ increases, $M$ relates better than $hr$ and $hw$ to the amount of contention generated by a superstep executed in bad mode.

# 5 Applications

In this section we check the behavior of our cost models on some application programs. Our goal here is to show that the Good and Bad cost functions described in the previous section provide a *prediction interval* of best-case and worst-case execution times for the communication in an arbitrary superstep. In other words, we check that the measured execution times do indeed lie between the times predicted by the `HrHwM-c` Good prediction and the `HrHwM` Bad prediction. We will also show that our prediction interval can be used as a tool for analyzing how well an application uses the memory hierarchy. For example, if the execution time approaches the Bad prediction, then the application is likely experiencing a substantial amount of congestion/contention in its interaction with the memory hierarchy.

The application programs implement three different sorting algorithms: sample sort [FM70, HC83, RV87, WS88], column sort [L85], and a parallel version of radix sort [BLM+91]. These algorithms were chosen because they are well-understood parallel algorithms that exhibit a variety of communication patterns.

## 5.1 Sorting algorithm implementations

All algorithms were coded in accordance with the programming model described in Section 3. No special effort was made to optimize the implementations since our goal was not to develop efficient sorting applications but rather to exercise our cost model on a variety of communication patterns arising in real applications. For simplicity, in the descriptions given below we assume that $p$, the number of processors, divides $n$, the number of elements to be sorted. Summaries of each algorithm's supersteps are shown in Tables 9, 10, and 11.

**Radix sort** Suppose each element is represented by $b$ bits. The simplest way to parallelize radix sort is to use a parallel version of counting sort (e.g., [CLR90, BLM+91]) as the internal stable sort. In `superstep 1`, each processor counts the number of its $n/p$ elements with each possible value in $[0, 2^r)$, where $r$ is the number of bits considered in each iteration. Next, the processors' counts are 'combined' by computing prefix sums of the processor-wise counts (`superstep 2`), and then adding an appropriate offset to these

values (`superstep 3`). Finally, in `superstep 4`, each processor places its elements in the output array. In our implementations, we used $r = 6$, so that the parallel counting sort is executed $\lceil b/r \rceil = \lceil 32/6 \rceil = 6$ times.

**Sample sort** There have been many sorting algorithms proposed that use a random sample of the input elements to partition the input into distinct subproblems that can be sorted independently (see, e.g., [FM70, HC83, RV87, WS88]). The basic version we implemented consists of 5 supersteps. In `superstep 1`, a random sample of $p-1$ *splitter* elements is selected from the $n$ input elements (defining $p$ buckets); we reduced the deviation in bucket sizes by selecting the splitters from a sample of size $100p$ (*oversampling*). In `superstep 2`, each processor counts the number of its $n/p$ input elements that fall in each bucket. These counts are processed in `superstep 3` to determine each processor's indices for each bucket. Next, the input elements are placed in the appropriate buckets (`superstep 4`), and finally, the elements in each bucket are sorted (`superstep 5`).

**Column sort** Column sort is a simple, elegant sorting algorithm. The input elements are placed in a two-dimensional array, and the method alternates between sorting columns of the array and permuting the array elements. There are four different permutations: transposing the matrix, 'reverse' transposing the matrix, right shifting the elements in the matrix (viewing it now as a one-dimensional matrix), and left shifting the elements in the matrix. To improve the efficiency of the sorting steps, we 'rotated' our matrix so that we were sorting rows rather than columns. To reduce the number of supersteps, we combined the 8 'natural' supersteps into 5 (see Table 11).

### 5.2 Experimental Results and Interpretation

We considered four processor configurations of the SGI PC, namely $p = 2, 4, 8$ and 12. On each configuration, we executed a large number of runs for each sorting algorithm covering a wide range of $n$, the number of integers to be sorted. In particular, we consider values $n \in \mathcal{N} = \mathcal{N}_0 \bigcup \mathcal{N}_1 \bigcup \mathcal{N}_2$, where $\mathcal{N}_0 = \{5000 * p * i \ : \ 1 \leq i \leq 10\}$, $\mathcal{N}_1 = \{50000 * p * i \ : \ 1 \leq i \leq 10\}$ and $\mathcal{N}_2 = \{550000 + 150000 * p * i \ : \ 0 \leq i \leq 9\}$. The input elements were random integers. For each run, the predicted times are obtained as the summation, over all supersteps of the run, of the predicted times for the copy-in and copy-out phases plus the measured time for the local computation phase (recall that our cost model is for communication only). The actual times reported are the summation of the actual times of all supersteps in the run.

**General analysis** For every algorithm, and for every value of $p$, the measured execution time did fall with the prediction interval given by our cost model, thus substantiating our claim that the `HrHwM-c` Good prediction and the `HrHwM` Bad prediction account for best-case and worst-case shared-memory access times, respectively. These results are shown in Figure 3. Each graph shows, for a fixed value of $p$, the measured execution time and either the Good or Bad predictions for all three sorting algorithms. For the `HrHwM` Bad predictions, we note that the relationship between the predicted and measured times for the various algorithms is invariant with $p$. However, for the `HrHwM-c` Good predictions, the measured times for radix and column sort (but not sample sort) tend away from the predicted times for large $n$ as $p$ grows (see discussion below).

It is interesting to note that in every case the relative order of the measured times for the algorithms matches the order of both the Good and Bad predictions. This implies that our cost model may prove valuable not only for establishing best-case and worst-case performance bounds for an application, but also for evaluating the relative performance of various algorithms.

In general, the closer an algorithm's measured time is to the Good prediction, the better its use of the memory hierarchy. Let $t_m$, $t_g$, and $t_b$, denote the measured time, the Good prediction, and the Bad prediction, respectively, for a given program or superstep. To quantify how close the measured time is to the Good predicted time, we consider the following two metrics:

**Loc**= $1 - (t_m - t_g)/(t_b - t_g)$. This metric is intended to measure the *degree of locality* of the program or superstep. For example, $Loc = 1$ if $t_m = t_g$, and $Loc = 0$ if $t_m = t_b$.

**M/G**= $t_m/t_g$. This metric is the ratio between the measured time and the Good prediction which may be more meaningful when the degree of locality is high.

Although it is difficult to tell from the graphs, due to the differing scales for the Good and Bad predictions, we found that for all sorting algorithms and for all values of $p$, the degree of locality as measured by *Loc* was always quite high (above 90%). Indeed, this is to be expected, as the Bad family access patterns exhibit a very high degree of contention and congestion that is unlikely to be experienced by a well-designed application program. In terms of the *M/G* ratio, the measured times for the algorithms ranged from around 2 times to about 8-10 times greater than the Good predicted time.

**Algorithm specific analysis** First, we note that the three programs did exhibit different ranges in their *M/G* factors: radix sort's ranged from about 2 for $p = 2$ to about 4.5 for $p = 12$, sample sort's remained relatively constant around 7-8 for all $p$, and column sort's ranged from about 5 for $p = 2$ to about 10 for $p = 12$. This would imply that radix sort is the most successful algorithm in terms of exploiting the memory hierarchy. Indeed, this may explain why radix sort is faster than both sample sort and column sort for small values of $n$ (which cannot be appreciated in the graphs due to the scale). We also note that sample sort's *M/G* behavior scales well with $p$, whereas both radix sort and column sort's *M/G* ratios tend to degrade. That is, as $p$ grows, radix and column sort's utilization of the memory hierarchy degrades. This is explained by the fact that these algorithms involve significantly more data movement than sample sort, and this becomes increasingly expensive as more processors become active in the system.

Summary analyses of the various supersteps in each sorting algorithm are given in Tables 9, 10, and 11 (the *M/G* ratio was only calculated for the 'significant' supersteps).

In columnsort, we note that although all supersteps have the same $(hr, hw, M)$ value, they do not all exhibit similar *Loc* or *M/G* measures. In particular, superstep 3 has a markedly lower (higher) *Loc* (*M/G*) value. This superstep performs the reverse-transpose operation in which elements are copied *backwards* in the arrays in question (our implementation stored each column in an array). This backwards traversal of the array will cause more invalidations than the permutations performed in the other supersteps, and is thus the likely cause of the reduced locality measures. Moreover, it can be seen that this superstep is the one responsible for column sort's degrading performance as $p$ grows. This is an example of how our cost model can be used to profile an application and determine which supersteps are in need of optimization.

# 6 Future Work

A number of aspects of our work deserve further investigation.

- Assessing the predictability of random access patterns and comparing the costs of such patterns against their counterparts in the good and bad families. This is of particular interest when the mapping between global variables and memory banks is done through hashing. (A suite of experiments targeting this goal is under development.)

- Verifying that our approach to cost modeling is not specific to the SGI PC but applies to other SMP machines. (We are in the process of porting our set of experiments to a Hewlett-Packard/Convex V-Class distributed shared-memory multiprocessor.)

- Trying to extend the applicability of our prediction model to programs which do not comply with the bulk-synchronous paradigm.

- Exploring further uses of our prediction interval in profiling and optimizing parallel software.

**Acknowledgement**

We would like to thank Lawrence Rauchwerger for useful advice. We would also like to thank Burchan Bayazit, who coded the initial version of column sort, and Hui Zheng, who helped with the superstepifying of sample sort.

## References

[ACS87] A. Aggarwal, A.K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. of the 28th IEEE Symp. on Foundations of Computer Science*, pages 204–216, 1987.

[BDM95] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: $c$-optimal multisearch for and extension of the BSP model. In *Proc. of the 3rd European Symposium on Algorithms*, pages 17–30, 1995.

[BHP+96] G. Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci and P. Spirakis. BSP vs. LogP. In *Proc. of the 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 25–32, June 1996.

[BGMZ97] G.E. Blelloch, P.B. Gibbons, Y. Matias, and M. Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 8(9):943–958, 1997.

[BLM+91] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Annual ACM Symp. Paral. Algor. Arch.*, pages 3–16, 1991.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[CKP+96] D.E. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K.E. Schauser, R. Subramonian, and T.V. Eicken. LogP: a practical model of computation. *Communications of the ACM*, 39(11):78–85, Nov 1996.

[DFRC93] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse-grained multicomputers. In *Proc. of the ACM Conference on Computational Geometry*, page 298–307, 1993.

[FM70] W. D. Frazer and A. C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, 1970.

[GHM+96] M. Goudreau, J.M.D. Hill, W. McColl, S. Rao, D.C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for the BSP worldwide standard library. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Rd., Oxford OX1 3QD, UK, 1996.

[GMR97] P.B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging-model for parallel computation? In *Proc. of the 9th ACM Symp. on Parallel Algorithms and Architectures*, pages 72–83, 1997.

[HP96] J.L. Hennessy and D.A. Patterson. *Computer Architecture A Quantitative Approach (Second Ed.)*. Morgan Kaufmann, San Francisco, CA, 1996.

[HC83] J. S. Huang and Y. C. Chow. Parallel sorting and data partitioning by sampling. In *Proceedings of the IEEE Computer Society's Seventh International Computer Software and Applications Conference*, pages 627–631, 1983.

[JW96] B.H.H. Juurlink and H.A.G. Wijshoff. A quantitative comparison of parallel computation models. In *Proc. of the 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 13–24, June 1996.

[L85] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Trans. Comput.*, c-34(4):344–354, 1985.

[Ncs97] NCSA webpage for SGI Power Challenge – architecture and configuration. http://www.ncsa.uiuc.edu/SCD/Hardware/PCA/Doc/Arch.html.

[RV87] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, 1987.

[Sil95] Silicon Graphics Corporation 1995. *SGI Power Challenge: User's Guide*, 1995.

[SRG94] J.P. Singh, E. Rothberg and A. Gupta. Modeling communication in parallel algorithms: a fruitful interaction between theory and systems? In *Proc. of the 6th ACM Symp. on Parallel Algorithms and Architectures*, pages 189–199, June 1994.

[Val90] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[WS88] Y. Won and S. Sahni. A balanced bin sort for hypercube multiprocessors. *Journal of Supercomputing*, 2:435–448, 1988.

| $p=8$ | | $L$ | $g_h$ | $g_{hr}$ | $g_{hw}$ | $g_M$ |
|---|---|---|---|---|---|---|
| | | | | $g_{hrm}, g_{hrc}$ | $g_{hwm}, g_{hwc}$ | |
| H | **R0** | 141 | 0.0236 | — | — | — |
| | **R1** | -33137 | 0.0897 | — | — | — |
| HrHw | **R0** | 138 | — | 0.0186 | 0.0090 | — |
| | **R1** | -34252 | — | 0.0573 | 0.0487 | — |
| HM | **R0** | 141 | 0.0180 | — | — | 0.00062 |
| | **R1** | -33137 | 0.0606 | — | — | 0.00322 |
| HrHwM-c | **R0** | 140 | — | 0.0184, 0.0 | 0.0087, 0.0 | 0.00005 |
| | **R1** | 0 | — | 0.0202, 0.0492 | 0.0082, 0.0444 | 0.00027 |

Table 1: Coefficients of all functions fit from data in Suite 1 for $p = 8$ processors (Good family).

| | | $L$ | $g_{hrc}$ | $g_{hrm}$ | $g_{hwc}$ | $g_{hwm}$ | $g_M$ |
|---|---|---|---|---|---|---|---|
| $p=2$ | **R0** | -30 | 0.0184 | 0.0 | 0.0086 | 0.0 | 0.00009 |
| | **R1** | -5153 | 0.0256 | 0.0482 | 0.0159 | 0.0442 | 0.00018 |
| $p=4$ | **R0** | 18 | 0.0185 | 0.0 | 0.0084 | 0.0 | 0.00013 |
| | **R1** | -2162 | 0.0230 | 0.0489 | 0.0113 | 0.0444 | 0.00024 |
| $p=8$ | **R0** | 140 | 0.0184 | 0.0 | 0.0087 | 0.0 | 0.00005 |
| | **R1** | 0 | 0.0202 | 0.0492 | 0.0082 | 0.0444 | 0.00027 |
| $p=12$ | **R0** | 279 | 0.0183 | 0.0 | 0.0087 | 0.0 | 0.00004 |
| | **R1** | 0 | 0.0191 | 0.0491 | 0.0081 | 0.0444 | 0.00035 |

Table 2: Coefficients of the function $\texttt{HrHwM-c}(hrc, hrm, hwc, hwm, M) = L + g_{hrc} \cdot hrc + g_{hrm} \cdot hrm + g_{hwc} \cdot hwc + g_{hwm} \cdot hwm + g_M \cdot M$ fit from data in Suite 1 (Good family).

| $p=8$ | $L$ | $g_h$ | $g_{hr}$ | $g_{hw}$ | $g_M$ |
|---|---|---|---|---|---|
| H | 13055 | 1.8974 | — | — | — |
| HrHw | 13767 | — | 0.7072 | 1.5467 | — |
| HM | 18791 | 0.3221 | — | — | 0.20423 |
| HrHwM | 16566 | — | 0.4612 | 0.7708 | 0.11138 |

Table 3: Coefficients of all functions fit from data in Suite 2 for $p = 8$ processors (Bad family).

| | $L$ | $g_{hr}$ | $g_{hw}$ | $g_M$ |
|---|---|---|---|---|
| $p=2$ | 10444 | 0.5062 | 1.0377 | 0.0484 |
| $p=4$ | 16261 | 0.4748 | 1.0166 | 0.0663 |
| $p=8$ | 16566 | 0.4612 | 0.7708 | 0.1113 |
| $p=12$ | 16555 | 0.2429 | 0.5755 | 0.1697 |

Table 4: Coefficients of the function $\texttt{HrHwM}(hr, hw, M) = L + g_{hr} \cdot hr + g_{hw} \cdot hw + g_M \cdot M$ fit from data in Suite 2 (Bad family).

| | | $p = 2$ | | $p = 4$ | | $p = 8$ | | $p = 12$ | |
|---|---|---|---|---|---|---|---|---|---|
| | | **Ave.** | **Max** | **Ave.** | **Max** | **Ave.** | **Max** | **Ave.** | **Max** |
| H | **R0** | 0.151 | 0.430 | 0.175 | 0.888 | 0.177 | 1.089 | 0.170 | 1.258 |
| | **R1** | 0.188 | 0.480 | 0.217 | 0.840 | 0.250 | 1.141 | 0.290 | 1.220 |
| HrHw | **R0** | 0.046 | 0.386 | 0.037 | 0.321 | 0.050 | 0.303 | 0.066 | 0.362 |
| | **R1** | 0.034 | 0.323 | 0.061 | 0.641 | 0.150 | 1.461 | 0.241 | 1.090 |
| HM | **R0** | 0.134 | 0.409 | 0.148 | 0.698 | 0.148 | 0.899 | 0.145 | 1.048 |
| | **R1** | 0.144 | 0.398 | 0.172 | 0.501 | 0.158 | 0.498 | 0.160 | 0.615 |
| HrHwM-c | **R0** | 0.046 | 0.388 | 0.036 | 0.310 | 0.049 | 0.296 | 0.065 | 0.360 |
| | **R1** | 0.014 | 0.077 | 0.017 | 0.116 | 0.026 | 0.136 | 0.049 | 0.262 |

Table 5: Validations for Suite 2 on functions fit from data in Suite 1 (Good family).

| | | $p = 2$ | | $p = 4$ | | $p = 8$ | | $p = 12$ | |
|---|---|---|---|---|---|---|---|---|---|
| | | **Ave.** | **Max** | **Ave.** | **Max** | **Ave.** | **Max** | **Ave.** | **Max** |
| H | **R0** | 0.089 | 0.603 | 0.087 | 0.480 | 0.088 | 0.444 | 0.089 | 0.450 |
| | **R1** | 0.074 | 0.270 | 0.096 | 0.248 | 0.111 | 0.365 | 0.141 | 0.800 |
| HrHw | **R0** | 0.048 | 0.566 | 0.044 | 0.489 | 0.054 | 0.444 | 0.070 | 0.435 |
| | **R1** | 0.026 | 0.133 | 0.055 | 0.243 | 0.115 | 0.311 | 0.198 | 0.452 |
| HM | **R0** | 0.074 | 0.566 | 0.068 | 0.470 | 0.080 | 0.446 | 0.087 | 0.426 |
| | **R1** | 0.068 | 0.260 | 0.074 | 0.258 | 0.078 | 0.344 | 0.093 | 0.445 |
| HrHwM-c | **R0** | 0.048 | 0.565 | 0.044 | 0.472 | 0.055 | 0.438 | 0.069 | 0.431 |
| | **R1** | 0.016 | 0.072 | 0.022 | 0.103 | 0.042 | 0.200 | 0.070 | 0.367 |

Table 6: Validations for Suite 3 on functions fit from data in Suite 1 (Good family).

| | $p = 2$ | | $p = 4$ | | $p = 8$ | | $p = 12$ | |
|---|---|---|---|---|---|---|---|---|
| | **Ave.** | **Max** | **Ave.** | **Max** | **Ave.** | **Max** | **Ave.** | **Max** |
| H | 0.203 | 0.309 | 0.211 | 0.700 | 0.286 | 1.366 | 0.354 | 2.020 |
| HrHw | 0.075 | 0.222 | 0.097 | 0.279 | 0.178 | 0.355 | 0.253 | 0.731 |
| HM | 0.083 | 0.148 | 0.099 | 0.299 | 0.104 | 0.366 | 0.108 | 0.442 |
| HrHwM | 0.057 | 0.206 | 0.058 | 0.205 | 0.063 | 0.201 | 0.060 | 0.277 |

Table 7: Validations for Suite 1 on functions fit from data in Suite 2 (Bad family).

| | $p = 2$ | | $p = 4$ | | $p = 8$ | | $p = 12$ | |
|---|---|---|---|---|---|---|---|---|
| | **Ave.** | **Max** | **Ave.** | **Max** | **Ave.** | **Max** | **Ave.** | **Max** |
| H | 0.097 | 0.282 | 0.143 | 0.386 | 0.145 | 0.4013 | 0.133 | 0.594 |
| HrHw | 0.051 | 0.358 | 0.078 | 0.451 | 0.084 | 0.290 | 0.090 | 0.460 |
| HM | 0.096 | 0.449 | 0.085 | 0.556 | 0.066 | 0.513 | 0.070 | 0.438 |
| HrHwM | 0.048 | 0.359 | 0.056 | 0.480 | 0.049 | 0.397 | 0.059 | 0.409 |

Table 8: Validations for Suite 3 on functions fit from data in Suite 2 (Bad family).

| Radix Sort Superstep Analysis | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Superstep** | | | $p = 2$ | | $p = 4$ | | $p = 8$ | | $p = 12$ | |
| SS# | description | $hr, hw, M$ | **Loc** | **M/G** | **Loc** | **M/G** | **Loc** | **M/G** | **Loc** | **M/G** |
| 1 | count elts | $\frac{n}{p}, 64, n + 64p$ | .97 | 1.65 | .96 | 1.84 | .95 | 2.78 | .96 | 3.76 |
| 2 | prefix sum | $64, 64, 128p$ | .99 | – | .99 | – | .97 | – | .95 | – |
| 3 | add offsets | $64, 64, 128p$ | .99 | – | .99 | – | .98 | – | .95 | – |
| 4 | move elts | $\frac{n}{p} + 64, \frac{n}{p}, 2n + 64p$ | .97 | 2.55 | .96 | 2.77 | .96 | 4.03 | .96 | 5.05 |

Table 9: Radix Sort Superstep Analysis. The *M/G* and *Loc* values (defined in the text) are averages. Good and Bad predictions are based on the functions `HrHwM-c` and `HrHwM`, respectively.

| Sample Sort Superstep Analysis | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Superstep** | | | $p = 2$ | | $p = 4$ | | $p = 8$ | | $p = 12$ | |
| SS# | description | $hr, hw, M$ | **Loc** | **M/G** | **Loc** | **M/G** | **Loc** | **M/G** | **Loc** | **M/G** |
| 1 | get sample | $100, 100, 200p$ | .96 | – | .97 | – | .95 | – | .91 | – |
| 2 | count elts | $\frac{n}{p} + p, p, n + 2p^2$ | .82 | 7.34 | .85 | 6.83 | .91 | 6.81 | .93 | 7.12 |
| 3 | prefix sum | $p, p, 2p^2$ | .99 | – | .99 | – | .97 | – | .95 | – |
| 4 | move elts | $\frac{n}{p} + 3p, \frac{n}{p}, 2n + 3p^2$ | .86 | 8.58 | .88 | 8.21 | .91 | 8.58 | .94 | 8.69 |
| 5 | sort bkts | $\frac{n}{p}, \frac{n}{p}, 2n$ **ideal** | .87 | 7.82 | .87 | 7.80 | .91 | 7.81 | .94 | 7.65 |

Table 10: Sample Sort Superstep Analysis. The *M/G* and *Loc* values (defined in the text) are averages. Good and Bad predictions are based on the functions `HrHwM-c` and `HrHwM`, respectively.

| Column Sort Superstep Analysis | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Superstep** | | | $p = 2$ | | $p = 4$ | | $p = 8$ | | $p = 12$ | |
| SS# | description | $hr, hw, M$ | **Loc** | **M/G** | **Loc** | **M/G** | **Loc** | **M/G** | **Loc** | **M/G** |
| 1 | init matrix | $\frac{n}{p}, \frac{n}{p}, 2n$ | .93 | 5.18 | .93 | 5.44 | .93 | 7.41 | .94 | 8.36 |
| 2 | sort/transp | $\frac{n}{p}, \frac{n}{p}, 2n$ | .95 | 3.62 | .96 | 3.72 | .95 | 4.61 | .96 | 6.16 |
| 3 | sort/rev-transp | $\frac{n}{p}, \frac{n}{p}, 2n$ | .85 | 9.97 | .81 | 13.34 | .78 | 19.69 | .76 | 31.09 |
| 4 | sort | $\frac{n}{p}, \frac{n}{p}, 2n$ | .95 | 3.89 | .95 | 4.06 | .95 | 5.26 | .95 | 6.62 |
| 5 | rshift/sort/lshift | $\frac{n}{p}, \frac{n}{p}, 2n$ | .95 | 3.72 | .96 | 3.70 | .96 | 4.45 | .96 | 5.52 |

Table 11: Column Sort Superstep Analysis. The *M/G* and *Loc* values (defined in the text) are averages. Good and Bad predictions are based on the functions `HrHwM-c` and `HrHwM`, respectively.

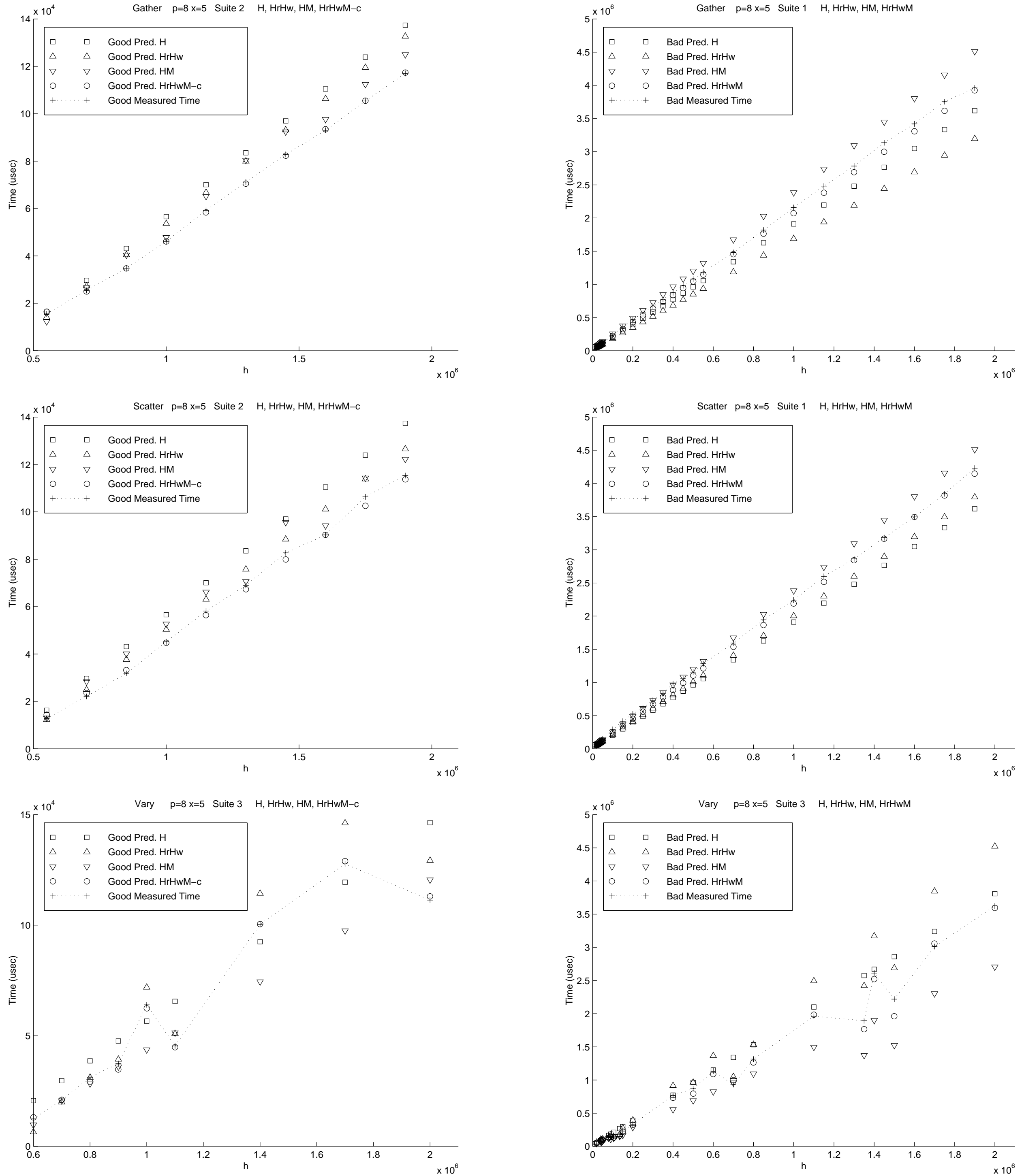


Figure 1: Cost functions: measured vs. predicted times for all functions. For each pattern (like-gather, like-scatter, vary), we show the Good and Bad measured and predicted times for $x = 5$, $p = 8$. The gather and scatter Good (Bad) case shows data from Suite 2 (Suite 1), and the vary Good and Bad cases show data from Suite 3. (Recall the good cost functions were fit using Suite 1 data and the bad cost functions were fit using Suite 2 data.) Note that in all cases, `HrHwM-c` and `HrHwM` are clearly the best predictors, while `H` is clearly less reliable. Similar results were observed for all patterns and Suites.

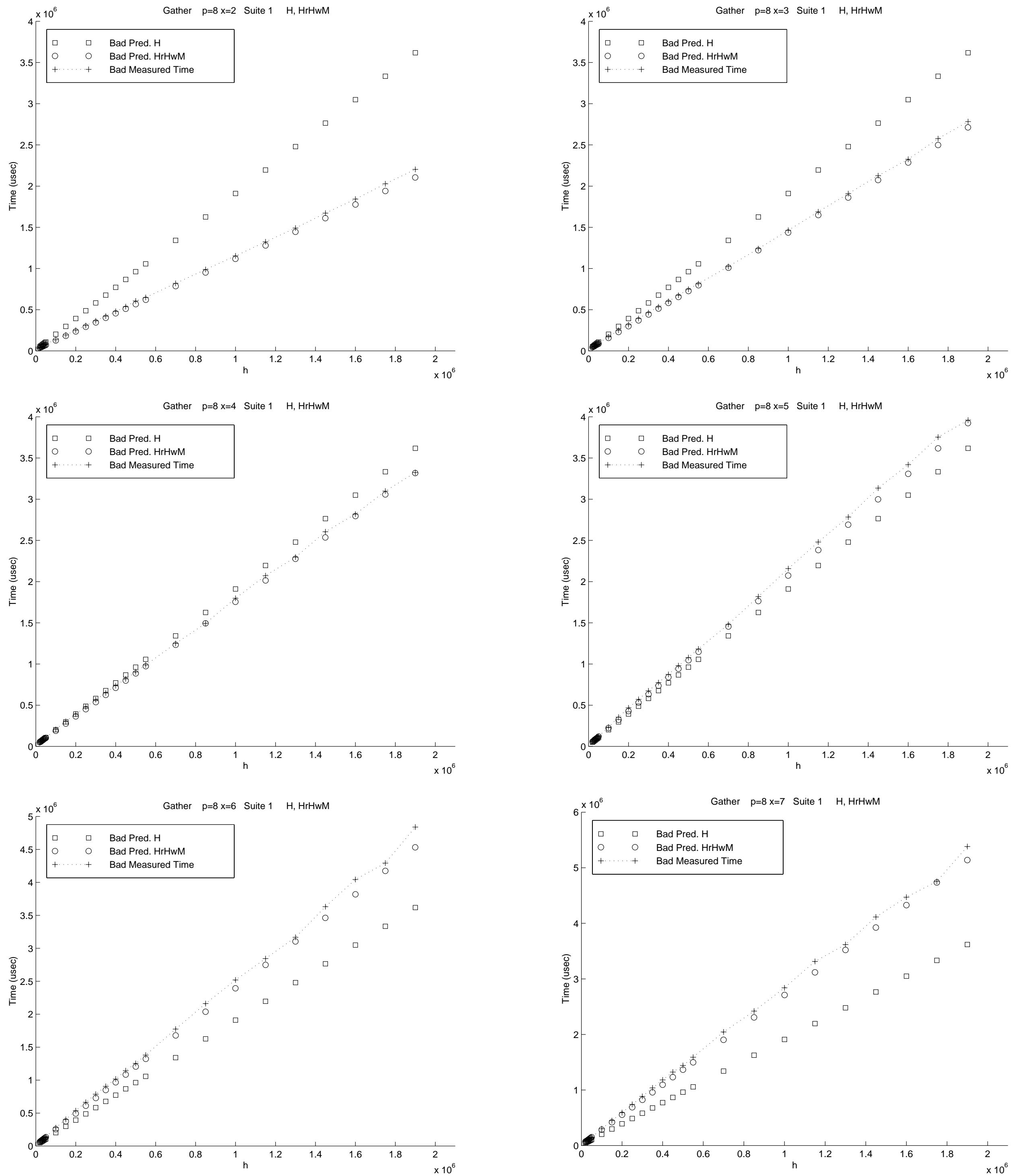


Figure 2: Cost functions: measured vs. predicted for `H` and `HrHwM`, like-gather patterns from Bad family, $x = 2, 3, 4, 5, 6, 7$, $p = 8$, Suite 1 data (cost functions fit from Suite 2 data). Notice how `HrHwM` predicts well the actual time, whereas `H` overestimates for smaller $x$ and underestimates for larger $x$. A similiar behavior was observed for the Good family, and for all other access patterns.

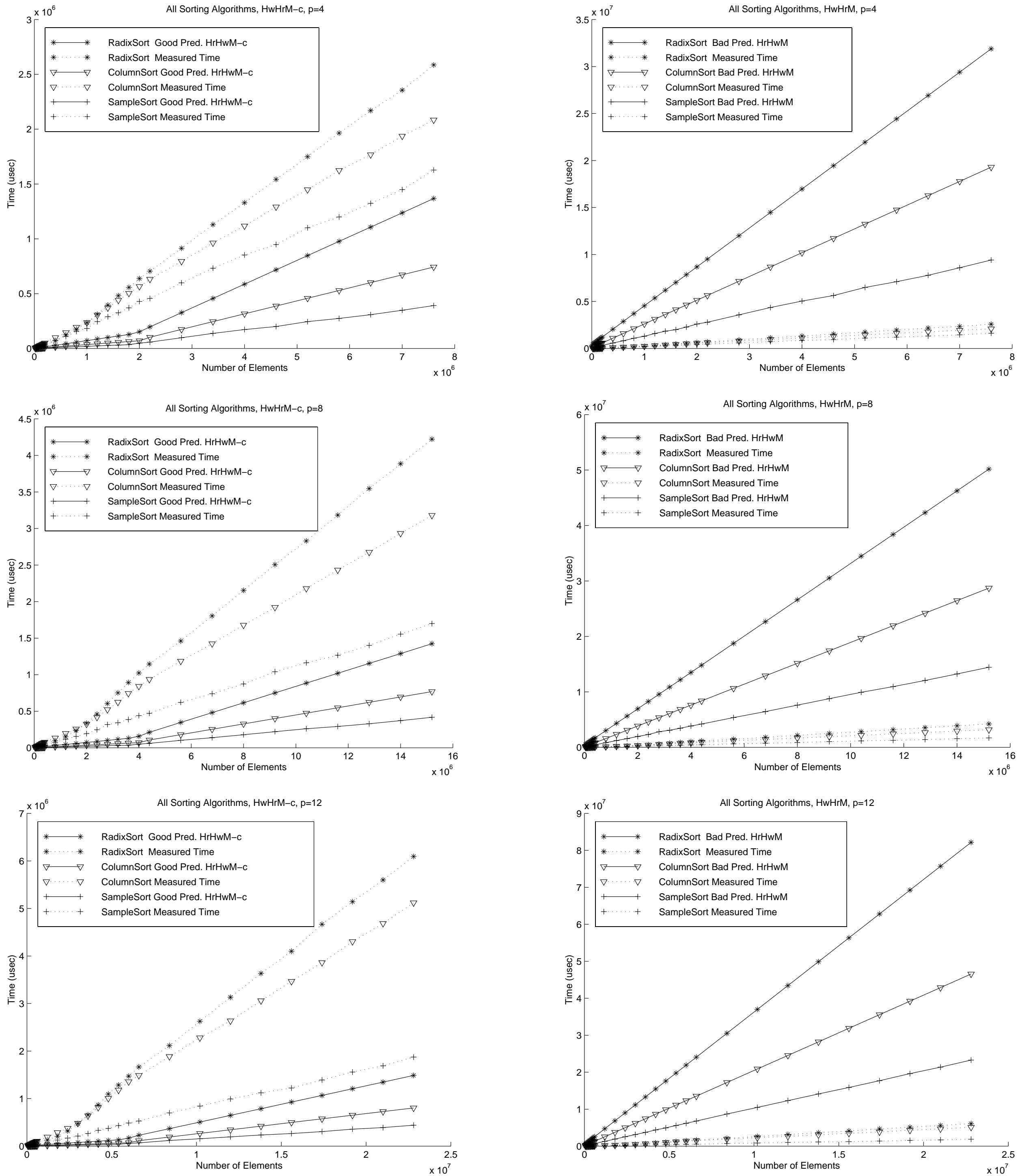


Figure 3: All Sorts: measured times vs. predicted times by `HrHwM-c` (Good) and `HrHwM` (Bad), $p = 4, 8, 12$. Notice how the distance between measured times and Good predictions diverges as $p$ grows, for radixsort and columnsort, whereas it remains relatively constant in the other cases.