DEVELOPING A COST MODEL FOR COMMUNICATION ON A SYMMETRIC MULTIPROCESSOR

A Thesis

by

JOHN KIMBAL PERDUE

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 1998

Major Subject: Computer Science

DEVELOPING A COST MODEL FOR COMMUNICATION ON A SYMMETRIC MULTIPROCESSOR

A Thesis

by

JOHN KIMBAL PERDUE

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

Nancy Amato
(Chair of Committee)

Lawrence Rauchwerger
(Member)

A.L. Narasimha Reddy
(Member)

Wei Zhao
(Head of Department)

December 1998

Major Subject: Computer Science

ABSTRACT

Developing a Cost Model for Communication
on a Symmetric MultiProcessor. (December 1998)
John Kimbal Perdue, B.S., Texas A&M University
Chair of Advisory Committee: Dr. Nancy Amato

Despite numerous advances in hardware design, the full potential of today's parallel processing systems is rarely realized due to the difficulty in predicting how a particular algorithm will utilize the underlying architecture. This study looks at a number of proposed "bridging models" for the bulk-synchronous parallel processing paradigm that attempt to provide an accurate abstraction of how well an algorithm will exploit the underlying architecture. The study provides evidence that temporal and spatial locality of memory accesses must be considered if a bridging model is to be useful. Additionally, this study presents a pair of prediction functions that serve as indicators for best and worst case performance on a particular class of systems, symmetric multiprocessors – in particular, the SGI Power Challenge.

To Mr. Carlson at Kingwood High School
for keeping the Apple II lab open for me long after
the final bell of the school day had rung and everybody
else had gone home.

ACKNOWLEDGMENTS

I would like to thank the members of my graduate committee for their guidance in this research. Additionally, I would like to thank some of my fellow students for their efforts. Special thanks to Lucia Dale for everything she has done – from showing me the ropes to helping with the presentation of the analysis and providing sanity checks along the way, her assistance is greatly appreciated. I would also like to thank Burchan Bayazit, who coded the initial version of column sort, Hui Zheng, who helped with the superstepifying of sample sort, and Mark Mathis, whose help with our new analysis tools will accelerate our research to the next level.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

CHAPTER I

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 often due to the inadequacies of the cost model employed to assess their complexity.

A. Previous Work

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. The most popular attempts to defining bridging models have been made by Valiant [1] and by Culler et al. [2] who introduced, respectively, the BSP and the LogP models. Both models abstract a parallel machine as a set of processors with local memory, communicating via message passing through an interconnection medium of limited bandwidth. Communication costs are expressed as simple linear functions based on few architectural parameters related to bandwidth and latency.

BSP is a bulk-synchronous model where computation is organized as a sequence of *supersteps* separated by barrier synchronizations, and processors operate asynchronously within each superstep. The large body of work this model has generated

---

The journal model is *IEEE Transactions on Automatic Control.*

has demonstrated its suitability for the development of portable software (see e.g., [3]). However, as has been often observed, the simple BSP cost model offers only a coarse level of predictiveness, since it disregards architectural features of real machines which may have a dramatic impact on performance [4, 5]. In contrast, LogP avoids the use of barriers and provides a lower-level programming model, which takes into account additional architectural features (e.g., start-up communication overhead) that enhance predictiveness but complicate algorithm analysis. (See [6] for a detailed comparison of BSP and LogP.)

Both BSP and LogP target distributed-memory architectures, where communication is realized via message passing. Two variations of the BSP model specifically tailored to shared-memory systems are the QSM [7] and the (d, x)-BSP [8], which both embody some aspects of memory contention. QSM is an abstract model which aims at highlighting some factors that influence asymptotic algorithm performance. In particular, its cost function includes a parameter that accounts for the maximum number of concurrent accesses to the same *memory location*, whose influence on performance can be appreciated only when the number of available processors, which is an upper bound to such a parameter, is large. Therefore, QSM predictions for machines with few processors tend to be inaccurate.

On the other hand, the (d, x)-BSP [8] aims at a higher level of descriptiveness for a certain class of shared-memory machines, characterized by numerous but slow memory banks, as is the case, for example, with vector multiprocessors. Its cost function includes a parameter that accounts for the maximum number of accesses to the same *memory bank* done in a superstep. However, bank contention may not be fully 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 on the run-time support.

B. Our Approach – Accounting for the Memory Hierarchy

None of the aforementioned models has been shown to reliably and accurately predict algorithmic performance on real machines. For shared-memory machines, we believe that improved accuracy can only be obtained by finding some way to account for the impact of the memory hierarchy on performance. Indeed, an application's interaction with the memory hierarchy is a crucial factor for Symmetric MultiProcessors (SMPs), a generation of machines characterized by off-the-shelf microprocessors communicating through a (distributed) shared memory via a shared communication network (e.g., a bus). In these machines, 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 cache (L2), to hundreds of cycles if data is brought from main memory. The cost may be even greater in the presence of invalidations or false-sharing. Unfortunately, however, none of the models mentioned above (QSM and (d, x)-BSP), account for the influence of the memory hierarchy on performance, except for the distinction between local and global memory accesses.

The objective of this work, is to bring in memory hierarchy effects when evaluating *algorithm communication costs* on SMPs. By focusing on a specific class of machines, we differ from the aforementioned previous modeling efforts, which were mainly concerned with proposing universal models of computation, thus sacrificing accuracy in the effort of achieving generality. We conduct our study on the SGI Power Challenge (SGI PC), which is a distinguished representative of the class of SMPs. For such a machine, we determine under which conditions *simple* cost functions can guarantee *accurate* predictions.

In an effort to analyze the effects of the memory hierarchy on shared memory accesses, we identify two extreme families of access patterns that maximize benefits

and penalties, respectively, incurred when interacting with the memory hierarchy. When then go on to show how this prediction interval between best and worst cases can be used as a model validator and an application profiler.

C. Hardware and Software Platforms

We conducted our study using the SGI PC as a representative platform for the class of SMPs. The PC system we used consists of RISC R10000 processors each provided with 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. Processors communicate through a 4-way interleaved, 4 GB distributed shared-memory, which is accessed via a 1.2GB/s shared-bus using a cache-coherent protocol [9, 10].

We adhered to an SPMD bulk-synchronous programming style [1, 7], where each processor executes the same program consisting of a sequence of *supersteps* separated by barriers. In a superstep, *global reads* and *writes* to the shared memory are confined to a copy-in and a copy-out phase, respectively, and are thus separated from local computation. In this fashion we are able to precisely account for the fraction of time spent in a superstep for communication, i.e, accesses to the shared memory.

On the SGI Power Challenge we implemented the above SPMD programming style by combining standard sequential C code with 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. The `m_sync` primitive implements a barrier and relies on a hardware synchronization mechanism whose cost is quite low, averaging less than 200 μs for 12 processors. In our programs, the supersteps correspond to `m_fork` calls, with their copy-in and a copy-out phases

separated from local computation by `m_sync`'s.

Each experiment was repeated at least three times; the time attributed to a superstep was the sum of the maximum time spent by any processor (perhaps different) in the copy-in and copy-out phases, which were timed separately in the parallel segment. Memory mapped counters on the SGI PC were used to obtain accurate times with little overhead. The machine was used in dedicated mode (no other users).

CHAPTER II

PREDICTING COMMUNICATION COSTS

The goal of our work is to understand how an application's interaction with the memory hierarchy impacts performance, and to develop a simple framework for obtaining useful predictions of communication costs. Our approach involved an extensive experimental component, and can be viewed as a higher-level counterpart of micro-benchmarking efforts (e.g., [11]). In particular, the methodology we applied can be outlined as follows:

1. Identify a small set of *parameters* believed to affect performance, and determine *best and worst usages* of the memory hierarchy.

2. Select *suites of access patterns* that exercise various combinations of these parameters and map them to the two usages.

3. Synthesize *cost functions* from these access patterns.

4. *Compare* the predictive quality of the functions separately for each usage.

A. Parameters and Best/Worst Memory Usages

For simplicity, we envision the shared memory as an array of integers. For an arbitrary superstep, let hr_i (resp., hw_i) denote the number of global reads (resp., writes) executed by the i-th processor, and let $hr = \max_i hr_i$, $hw = \max_i hw_i$ and $M = \sum_i (hr_i + hw_i)$. Thus M represents the total communication volume of the superstep. Let also $h = \max\{hr, hw\}$. The multiset of values $\{(hr_i, hw_i) \ : \ 1 \leq i \leq p\}$ defines the *access pattern* realized by the superstep.

As indicated by practice, and as will be further confirmed by our experiments, the

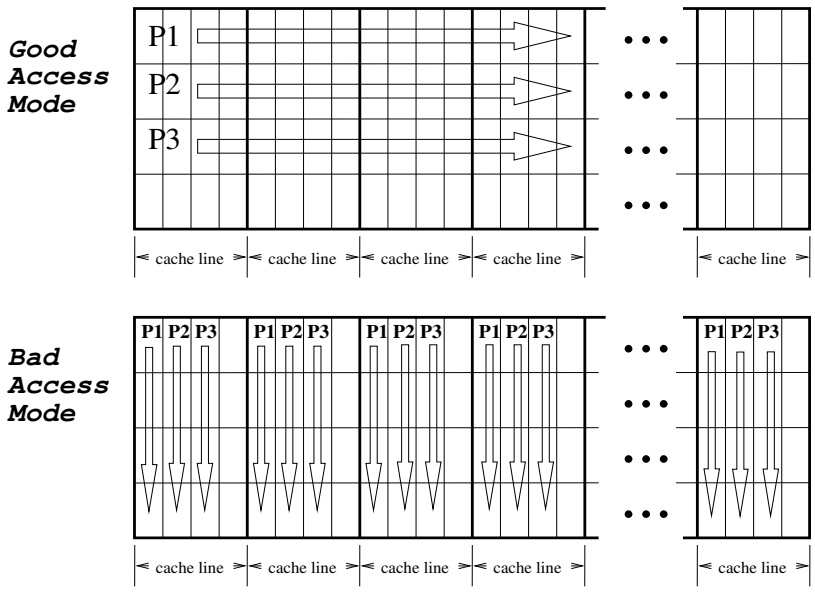

Fig. 1. Accesses performed by processors P_i, $i = 1, 2, \ldots$, in good and bad supersteps.

interaction with the memory hierarchy can dramatically affect the cost of an access pattern, even when all other parameters are constant. The impact of the hierarchy is related to a number of phenomena that are hard to capture in a quantitative fashion, namely, the *locality of reference* of the actual accesses performed, which in turn relates to the level of the hierarchy being accessed and to the contiguity of consecutive accesses, and on the *write conflicts* among the processors, which activate the invalidation mechanism of the cache coherence protocol. Our first goal is to show that the cost of an access pattern can be accurately predicted by a linear function of the three parameters (hr, hw, M), as long as the influence of the above phenomena is fixed.

We consider two extreme families of supersteps, referred to as *Good* and *Bad*, where accesses to the shared memory are realized in such a way as to maximize benefits and penalties, respectively, caused by the interaction with the memory hierarchy. Specifically, consider a superstep realizing an access pattern $\{(hr_i, hw_i) \ : \ 1 \leq i \leq p\}$.

We say that the superstep belongs to the Good family (*good superstep*) if Processor i reads (resp., writes) the (consecutive) integers stored in the array locations with indices $(i-1)t+k$, for $0 \leq k < hr_i$ (resp., $0 \leq k < hw_i$), where t is large enough so that processors access non-overlapping memory chunks. Additionally, 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, we say that the superstep belongs to the Bad family (*bad superstep*) if Processor i reads (resp., writes) the array locations with indices $i + kt_{\text{line}}$, for $0 \leq k < hr_i$ (resp., $0 \leq k < hw_i$), where t_{line} denotes the number of integers that fit in a cache line (32 for L2 cache on our SGI PC). 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 for the cache lines.

A pictorial representation of the memory usage in good and bad supersteps is shown in Figure 1, and pseudo-code for the good and bad supersteps is shown in Figure 2.

B. Suites of Access Patterns

We considered four processor configurations of the SGI PC, namely $p = 2, 4, 8$ and 12, and performed a number of experiments for every configuration, exercising a large spectrum of access patterns. Each experiment consists of the execution of either a good or a bad superstep. 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

```
void Good-Superstep (Pattern, A)
{
  int id, hr, hw, gstride, temp;
  id = get_my_id();
  hr = Pattern[id].hr;
  hw = Pattern[id].hw;
  gstride = Pattern[id].gstride

 /* copy-in phase */
  prime_cache(A,id,hr,gstride);
  temp = 0;
  for (i=0; i<hr; i++)
    temp += A[id*gstride + i];

  m_sync();

 /* copy-out phase */
  prime_cache(A,id,hw,gstride);
  for (i=0; i<hw; i++)
    A[id*gstride + i] = temp;
}
```

```
void Bad-Superstep (Pattern, A)
{
  int id, hr, hw, bstride, temp;
  id = get_my_id();
  hr = Pattern[id].hr;
  hw = Pattern[id].hw;
  bstride = Pattern[id].bstride

 /* copy-in phase */
  flush_cache(A,id,hr,bstride);
  temp = 0;
  for (i=0; i<hr; i++)
    temp += A[i*bstride + id];

  m_sync();

 /* copy-out phase */
  flush_cache(A,id,hw,bstride);
  for (i=0; i<hw; i++)
    A[i*bstride + id] = temp;
}
```

Fig. 2. Pseudo-code for good and bad supersteps. The `for` loops in the copy-in and copy-out phases were timed separately on each processor. The time of the superstep was the sum of the maximum copy-in and copy-out times taken by any processor.

$\mathcal{H} = \mathcal{H}_0 \bigcup \mathcal{H}_1 \bigcup \mathcal{H}_2$.

Suite 1 For every $h \in \mathcal{H}$, $1 \leq x \leq p$ and for each of the following three access patterns, one good and one bad superstep realizing the pattern are executed:

`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` concentrates reads into a few processors, `like-scatter` concentrates writes into a few processors, and `vary` balances reads and writes while varying, as x grows, the overall number of accesses done by all processors. Note that when $x = p$ the three patterns coincide. In this case, we execute only one of them.

Suite 2 For every $h \in \mathcal{H}$ and $1 \leq x \leq p$, three good and three bad supersteps are executed realizing access patterns where the number of reads and writes performed by each processor is randomly chosen enforcing that the maximum number of reads (resp., writes) performed by any processor is equal to hr (resp., hw) of `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 (hr and hw) exercised in Suite 1, but randomize the selection of the specific access pattern and the overall communication volume (M).

Suite 3 For every $h \in \mathcal{H}$ and $1 \leq x \leq p$, three good and three bad supersteps are executed realizing access patterns where the number of reads and writes performed by each processor is randomly chosen enforcing 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 (M) rather than maximum number of reads and writes (hr and hw).

C. Cost Functions

For each family of supersteps, we investigated 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.

For each family, we examined the following cost functions:

$$\begin{aligned}
\texttt{H}(h) &= L + g_h \cdot h \\
\texttt{HM}(h, M) &= L + g_h \cdot h + g_M \cdot M \\
\texttt{HrHw}(hr, hw) &= L + g_{hr} \cdot hr + g_{hw} \cdot hw \\
\texttt{HrHwM}(hr, hw, M) &= L + g_{hr} \cdot hr + g_{hw} \cdot hw + g_M \cdot M \text{ (Bad Family)} \\
\texttt{HrHwM-c}(hr, hw, M) &= L + g_{hrc} \cdot hrc + g_{hrm} \cdot hrm + \\
&\quad g_{hwc} \cdot hwc + g_{hwm} \cdot hwm + g_M \cdot M \text{ (Good Family)}
\end{aligned}$$

Function `HrHwM-c` is a slight modification of `HrHwM`, obtained by splitting the contributions of hr and hw into two terms, namely hrc and hrm for hr, and hwc and hwm for hw. In particular, we let $hrc = \min(hr, |L2|)$ and $hrm = hr - hrc$, where $|L2|$ is the size of the L2 cache. Similarly, we let $hwc = \min(hw, |L2|)$ and $hwm = hw - hwc$. In this way, we can model cache benefits for the first $|L2|$ accesses. This partitioning resulted in more accurate predictions for the Good family. In contrast, for the Bad family it was unnecessary to distinguish between cache accesses and memory accesses in function `HrHwM`, since the Bad family comprises supersteps that do not take any significant advantage of the cache.

In order to improve predictiveness, we subdivided 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. $R0$ contains all supersteps with $h \leq |L2|$ (which all have $hrm = hwm = 0$) and $R1$ contains those with $h > |L2|$. No subdivision was necessary for the Bad family.

For the Good family, the functions have been obtained from Suite 1 data, and their predictiveness has been assessed on Suite 2 and Suite 3 data. 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` (the most accurate function) for all processor configurations (see Table 2). (Note that the cost functions express running times in μs.) The average and maximum relative errors between actual and predicted times for the supersteps of Suites 2 and 3, are reported in Tables 3 and 4 for all functions

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

$p = 8$		L	g_h	g_{hr}	g_{hw}	g_M
				g_{hrc}, g_{hrm}	g_{hwc}, g_{hwm}	
`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 2. Coefficients of the function `HrHwM-c`$(hr, hw, M) = L + g_{hrc} \cdot hrc + g_{hrm} \cdot hrm + g_{hwc} \cdot hwc + g_{hwm} \cdot hwm + g_M \cdot M$ fit from data in Suite 1 (Good family).

`HrHwM-c`		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 3. Validations for Suite 2 on functions fit from data in Suite 1 (Good family).

Relative Errors		$p=2$		$p=4$		$p=8$		$p=12$	
		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

and all values of p.

For the Bad family, the functions have been obtained by fitting Suite 2 data, and their predictiveness has been assessed on Suite 1 and 3 data. In Table 5 we report the coefficients of all functions for $p = 8$, while in Table 6 we report the coefficients of `HrHwM` (the most accurate function) for all processor configurations. 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.

D. Comparisons

A number of observations concerning the cost functions and their estimated predictiveness are in order. In general, we note that not all functions exhibit the same predictive quality, although average errors never exceed 35%. Graphs showing pre-

Table 4. Validations for Suite 3 on functions fit from data in Suite 1 (Good family).

Relative Errors		$p=2$		$p=4$		$p=8$		$p=12$	
		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 5. Coefficients of all functions fit from data in Suite 2 for $p = 8$ processors (Bad 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 6. 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).

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

Relative	$p = 2$		$p = 4$		$p = 8$		$p = 12$	
Errors	**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 8. Validations for Suite 3 on functions fit from data in Suite 2 (Bad family).

Relative	$p = 2$		$p = 4$		$p = 8$		$p = 12$	
Errors	**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

dicted against measured times for a sample of patterns from both Good and Bad families are reported in Figure 3 on p. 17. In particular, it is easy to see that function `H`, which is analogous to the BSP cost function, exhibits the poorest predictiveness (e.g., see the graphs in Figure 4 on p. 18). 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.

This validates our hypothesis that when the impact of the memory hierarchy is fixed, as is the case for the two families of access patterns considered, the cost of an access pattern 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 here for brevity, 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.

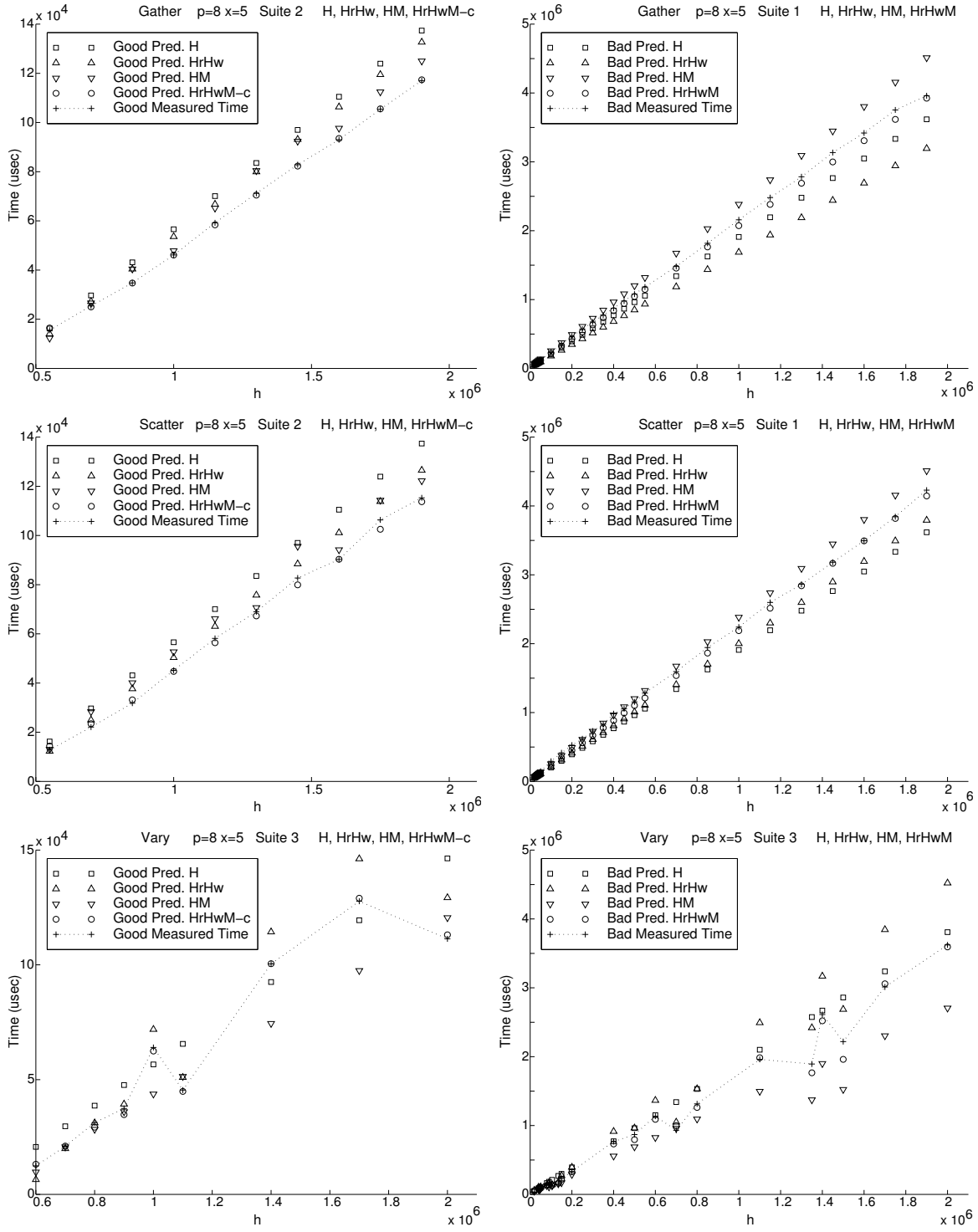

Fig. 3. Cost functions: measured vs. predicted times for all functions. protect For each pattern, we show the Good and Bad measured and predicted times for $x = 5$, $p = 8$. The gather and scatter Good (Bad) case shows data from Suite 2 (Suite 1), and the vary Good and Bad cases show data from Suite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

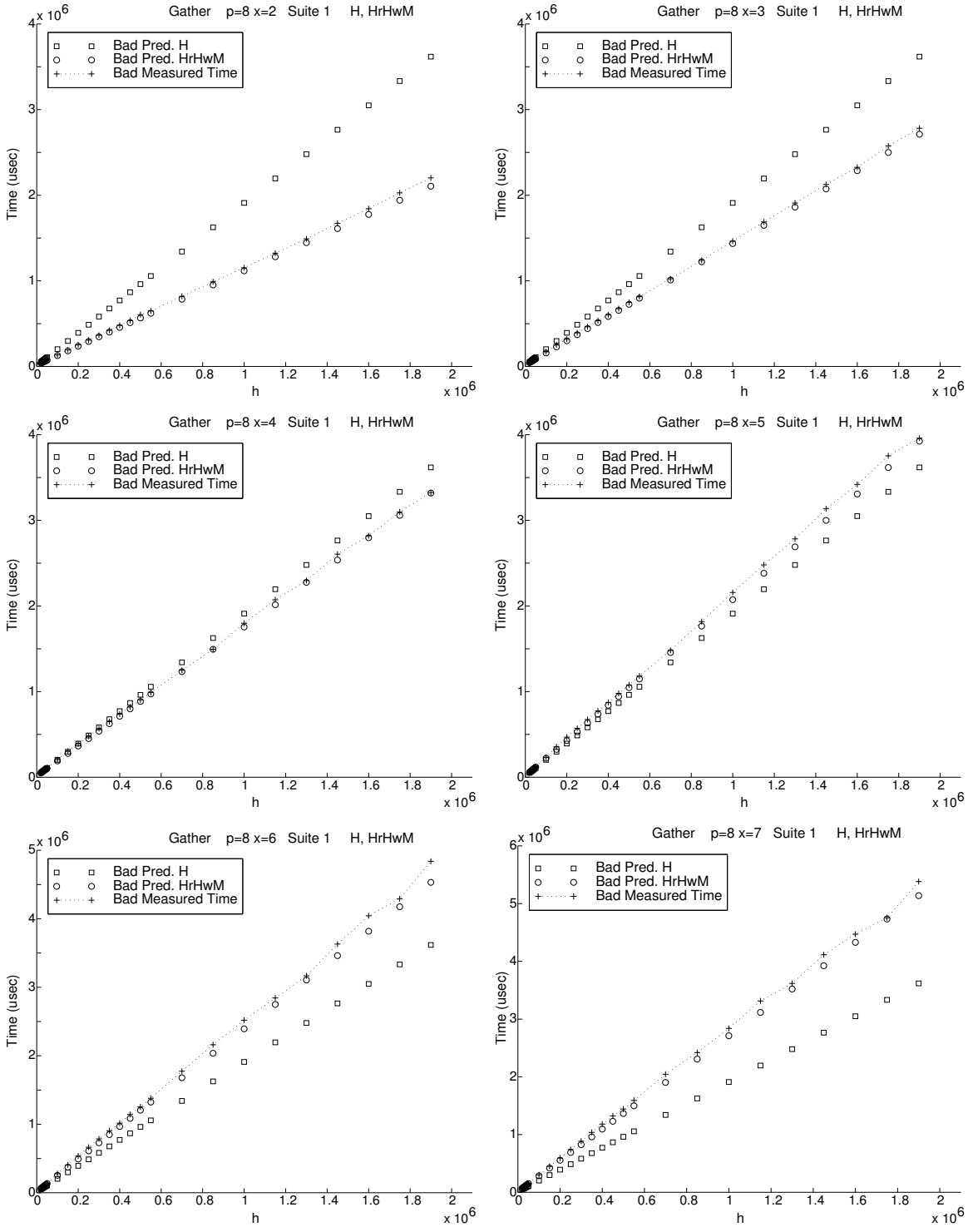

Fig. 4. Cost functions: measured vs. predicted for `H` and `HrHwM`, like-gather patterns from Bad family, $x = 2, 3, 4, 5, 6, 7$, $p = 8$, Suite 1 data (cost functions fit from Suite 2 data). Notice how `HrHwM` predicts well the actual time, whereas `H` overestimates for smaller x and underestimates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

Good family A number of conclusions can be drawn by looking at the coefficients of the `HrHwM-c` function (see Tables 1 and 2.) 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 are blocking while writes are not. 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%.

In summary, by looking at the validations of the different functions (see Tables 3 and 4), 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 fitted for the Bad family, with particular reference to `HrHwM`, whose predictive quality is definitively superior to that of the other functions (see Tables 5 and 6). 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 invalidations take place during the execution of a bad superstep. 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 (see Tables 7 and 8). 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.

CHAPTER III

PREDICTION INTERVAL – VALIDATION AND PROFILING TOOL

In order to support the validity of our framework and exercise the prediction interval, we performed two types of tests on the SGI PC: (1) we ran several supersteps realizing communication patterns of different degrees of locality; and (2) we ran three different sorting algorithms on several input sizes. In both cases, we verified that the measured communication costs always fall within the prediction interval, and tried to obtain an estimate of how well the application uses the memory hierarchy (e.g., degree of locality and contention) by means of the quantities $Loc = (t_b - t_m)/(t_b - t_g)$ and $M/G = t_m/t_g$, where, for a given program or superstep, t_m, t_g and t_b denote measured communication time, and Good and Bad predictions, respectively. The Loc value is an indicator of the distance of the measured time from the good and bad predictions, specifically $0 \leq Loc \leq 1$ and $Loc = 1$ when $t_m = t_g$, while $Loc = 0$ when $t_m = t_b$. Since the supersteps in the bad family represent a very extreme worst-case scenario, which is unlikely to be experienced by most applications, we introduced the M/G quantity, which helps distinguish patterns characterized by high Loc values.

A. Arbitrary Communication Patterns

We tried to assess the different contributions of spatial and temporal locality by running two sets of communication patterns with varying values of hr, hw and M. In the first set, each processors accesses distinct random blocks of data from the shared memory, where the block size equals the L2 cache line size. No such block initially resides in the processor's caches. In this fashion, spatial locality is preserved, while temporal locality is lost. In the second set, processors access completely random memory locations, so that both types of locality are lost. These latter patterns

closely resemble those in the Bad family, except that no attempt is made to maximize contention through false sharing.

The measured times of all the tested patterns fall within the prediction interval. Moreover, as expected, patterns in the first set exhibit a much better performance. In particular their average *Loc* value is always above 0.9, while the M/G ratio is consistently below 5. The (mild) performance loss with respect to patterns in the Good family is mainly due to lack of temporal locality. In contrast, the average *Loc* value of the patterns in the second set is very close to zero, a performance practically indistinguishable from that of the Bad family. These findings support two conclusions: preserving spatial locality is crucial for approaching best-case performance, and false sharing has less impact on performance than loss of locality.

B. Sorting Algorithms

In order to exercise the prediction interval given by our cost model on real applications, we implemented three different sorting algorithms: *samplesort* [12], *columnsort* [13], and a parallel version of *radixsort* [14]. We performed an extensive set of runs covering wide ranges of n, the number of random integers sorted. Since we model communication costs only, we limited ourselves to comparing measured versus predicted cumulative time of the copy-in/copy-out phases of the supersteps executed by each run.

As expected, the measured execution times always fall within the prediction interval. Moreover, in every case the relative order of the measured times for the algorithms is the same as their relative order in both the Good and Bad predictions (see Figure 5 on p. 23.) This suggests that our cost model may be profitably employed to forecast the relative performance of different algorithms for the same problem.

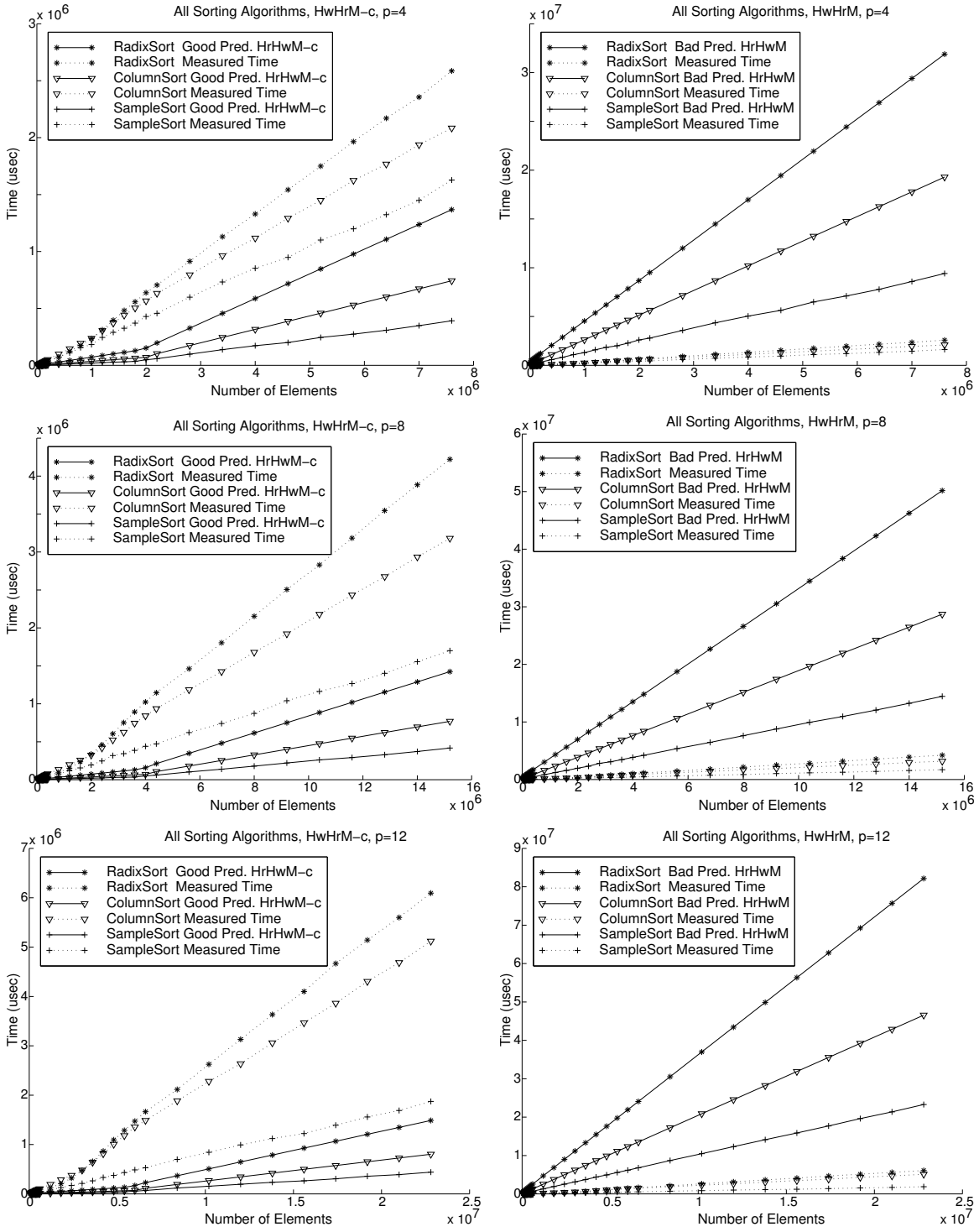

Fig. 5. All sorts: measured times vs. predicted times by `HrHwM-c` (Good) and `HrHwM` (Bad), $p = 4, 8, 12$. Notice how the distance between measured times and Good predictions diverges as p grows, for radixsort and columnsort, whereas it remains relatively constant in the other cases.

Table 9. 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.

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

We found that all sorting algorithms exhibit a fairly good use of memory hierarchy (average $Loc >= 0.9$). This is not surprising, as observed before, the Bad prediction assumes no locality and a very high degree of contention that are unlikely to be experienced by a well-designed application.

However, there are differences between the programs in terms of their M/G ratio. In particular, we noted that samplesort's utilization of the memory hierarchy scales better with p than those of the other algorithms. Indeed, radix and columnsort require more global data movement than samplesort and thus become increasingly more inefficient as more processors become active in the system.

The use of the prediction interval, and the resulting M/G values, as a profiling tool can be illustrated by examining the supersteps of columnsort (see Table 9 on p. 24 for a summary analysis of each superstep for $p = 2, 4, 8, 12$). 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, which reveals 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.

Table 10. 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.

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 11. 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.

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

CHAPTER IV

CONCLUSION

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.

- *Accurate predictions can be attained for families of access patterns that make a similar use of the hierarchy.* 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 function of few parameters related to mere access counts. Additionally, for each family, we compare the predictions of several cost functions to assess the relative importance of a number of parameters. We provide evidence that in order to achieve high accuracy, it is necessary to account separately for reads and writes, and to include a measure of the overall communication volume.

- *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) profile an application to gain valuable insights into its interaction with the memory hierarchy, which may in turn lead to better algorithm design [15].

Our conclusions are based on an extensive set of experiments. Specifically, we measure actual costs of three different suites of access patterns for each family, and use one suite to interpolate several cost functions, and the others to assess their

predictive quality. 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 a variety of communication patterns some of which are randomly generated while others arise in three different sorting algorithms. We verify that measured costs of these patterns always fall within the prediction interval, which provides evidence that the two families do indeed make extreme uses of the memory hierarchy. In addition, we illustrate the use of the prediction interval as a profiling tool to assess the influence of the memory hierarchy on the performance of the sorting algorithms.

REFERENCES

[1] L. Valiant, "Bridging model for parallel computation," *Comm. ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[2] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. E. Schauser, R. Subramonian, and T. V. Eicken, "LogP: A practical model of computation," *Comm. ACM*, vol. 39, no. 11, pp. 78–85, 1996.

[3] M. Goudreau, J. M. D. Hil, W. McColl, S. Rao, D. C. Stefanescu, T. Suel, and T. Tsantilas, "A proposal for the BSP worldwide standard library," Tech. Rep., Oxford University Computing Laboratory, 1996.

[4] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide, "Truly efficient parallel algorithms: c-optimal multisearch for an extension of the bsp model," in *Proceedings 3rd European Symposium on Algorithms*, 1995, pp. 17–30.

[5] B. H. H. Juurlink and H. A. G. Wijshoff, "A quantitative comparison of parallel computation models," in *Proc. ACM Symp. Par. Alg. Arch. (SPAA)*, 1996, pp. 13–24.

[6] G. Bilardi, K. T. Herley, A. Pietracaprina, Geppino Pucci, and P. Spirakis, "BSP vs. LogP," in *Proc. ACM Symp. Par. Alg. Arch. (SPAA)*, 1996, pp. 25–32.

[7] P. B. Gibbons, Y. Matias, and V. Ramachandran, "Can a shared-memory model serve as a bridging-model for parallel computation?," in *Proc. ACM Symp. Par. Alg. Arch. (SPAA)*, 1997, pp. 72–83.

[8] G.E. Blelloch, P.B. Gibbons, Y. Mattias, and M. Zagha, "Accounting for memory bank contention and delay in high-bandwidth multiprocessors," *IEEE Trans. Par. Dist. Sys.*, vol. 8, no. 9, pp. 943–958, 1997.

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

[10] NCSA, http://www.ncsa.uiuc.edu/SCD/Hardware/PCA/Doc/Arch.html, *NCSA webpage for SGI Power Challenge – Architecture and Configuration.*

[11] G.A. Abandah and E.S. Davidson, "Characterizing distributed shared memory performance: A case study of the Convex SPP1000," *IEEE Trans. Par. Dist. Sys.*, vol. 9, no. 2, pp. 206–216, 1998.

[12] Y. Won and S. Sahni, "A balanced bin sort for hypercube multiprocessors," *J. Supercomput.*, vol. 2, pp. 435–448, 1988.

[13] T. Leighton, "Tight bounds on the complexity of parallel sorting," *IEEE Trans. Comput.*, vol. c-34, no. 4, pp. 344–354, 1985.

[14] 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 *Proc. ACM Symp. Par. Alg. Arch. (SPAA)*, 1991, pp. 3–16.

[15] J. P. Singh, E. Rothberg, and A. Gupta, "Modeling communication in parallel algorithms: A fruitful interaction between theory and systems?," in *Proc. ACM Symp. Par. Alg. Arch. (SPAA)*, 1994, pp. 189–199.

VITA

John Kimbal Perdue was born April 11th, 1964, in Big Spring, Texas to John Robert and Sandra Rider Perdue. He graduated from Kingwood High School in 1982. He received his B.S. in Computer Science at Texas A&M University in May of 1996. Between 1982 and 1996, he worked in a number of computer-related positions. Mr. Perdue's permanent address is 708 Inlow Blvd., College Station, TX, 77840. He can also be reached via e-mail at j-perdue@tamu.edu. More information on his past and present work can be found at his home page available at http://www.cs.tamu.edu/people/jkp2866/.