CS 388P Write up

Jesus Gonzalez, Tomas McCandless

October 9, 2012

1 Summary of Contributions

1.1 A Bridging Model for Parallel Computation [3]

The author of this paper notes that the von Neumann model has traditionally served as a bridging model for sequential computation in the sense that it enables hardware designers to share a common goal, and it allows software designers to write efficient programs without explicit consideration of the underlying hardware. Similarly, a bridging model for parallel computation is necessary if we wish for general purpose parallel computation to succeed. To this end, the author introduces the BSP (bulk-synchronous parallel) model as a potential candidate for a parallel bridging model. The BSP contains a number of processor/memory components, a router that delivers messages between pairs of components, and a facility for synchronization.

1.2 A General Purpose Shared Memory Model for Parallel Computation [4]

The large amount of diversity in parallel architectures has made the design of a general purpose parallel model an important yet challenging task. In the past, designers have generally chosen to either design algorithms for the PRAM, or to design for a distributed memory architecture, modifying the algorithm as necessary to account for the topology of the network. The former approach ignores bandwidth limitations of machines, while the latter approach yields algorithms that are designed for a specific network topology rather than a general network. This paper defines the QSM model, presents

some algorithmic results for well known problems, and presents a work-preserving emulation of the QSM on the BSP.

1.3 What Good are Shared Memory Models? [5]

Shared memory models have been useful in the past, but they have the shortcoming of failing to model bandwidth limitations of machines. The QSM model accounts for bandwidth limitations while providing a shared memory abstraction. This is important in part because a shared memory abstraction is easier to use than a distributed memory or message passing architecture. This paper provides a fairly nontechnical overview of recent developments, discusses the attractive features of the QSM model as a candidate for a bridging model, and shows that the QSM can serve as an adequate model for designing algorithms on certain Cray machines.

1.4 Can a Shared-Memory Model Serve as a Bridging Model for Parallel Computation? [1]

A bridging model for parallel computation is a model that "spans the range from algorithm design to architecture to hardware". [1] Such a model should provide an abstraction that is easy for algorithm designers to use, and should be realizable by hardware and system software at varying price/performance points. The new model this paper introduces, the QSM, is meant to be an attractive candidate for a bridging model. The QSM consists of processors with private individual memory in addition to a global shared memory. The authors give a work preserving emulation (an emulation is work-preserving if

both models perform the same amount of work to within a constant factor) of the QSM on the BSP. However, since the QSM has fewer parameters than the BSP and does not deal directly with memory partitioning, we would expect it to be easier to design algorithms on the QSM. Designs can then be mapped onto the BSP with modest slowdown.

1.5 Emulations between QSM, BSP and LogP: A framework for general-purpose parallel algorithm design. [2]

The authors present work-preserving emulations with just a small slowdown between the general-purpose models: LogP, BSP and QSM. The relevance of this lies on the simplicity of the QSM model, which makes it easy to program for and, yet, through emulation on more realistic models like BSP and LogP run an algorithm with just a logarithmic slowdown.

The authors present 3 algorithms for basic problems (prefix sums, sample sort and list ranking). All of them are optimal and simulations show that the predicted performance for each of them are very close with just insignificant discrepancy.

2 Summary of Key Papers

2.1 Can a Shared-Memory Model Serve as a Bridging Model for Parallel Computation?

A bridging model for parallel computation is a model that "spans the range from algorithm design to architecture to hardware". [1] Such a model should provide an abstraction that is easy for algorithm designers to use, and should be realizable by hardware and system software at varying price/performance points.

Models of parallel computation should attempt to satisfy two goals that are in conflict: a model should be sufficiently abstract to allow algorithm designers to write programs that are simple and portable across architectures, and a model should also expose some low-level architectural details to allow for optimization. The PRAM model is both widely used and simple, yet it has been criticized for being too high-level and thus failing to accurately model parallel machines. Specifically, the PRAM does not model realities of current parallel machines, such as bandwidth limitations. Similarly, network-based models such as the hypercube have been criticized for being too low-level, failing to be widely reflective of the current technological state of parallel machines.

These shortcomings of existing models have led to the introduction of more intermediate models such as the BSP or LogP models.

The new model this paper introduces, the QSM, is meant to be an attractive candidate for a bridging model. The QSM consists of processors

with private individual memory in addition to a global shared memory.

One goal for a higher-level shared-memory model should be allowing efficient emulation on lower-level models which may be more realistic.

It is mistaken to view "shared-memory model" as being synonymous with PRAM. The shared-memory abstraction refers to interprocessor communication. That is, a processor may have additional memory as part of its local state. While the PRAM does not model bandwidth limitations, a shared-memory model may be asynchronous, or have explicit charges for communication.

One of the primary motivations for positing a shared-memory model as a bridging model is the portability across memory architectures that accompanies such models. This is because the layout of memory is hidden by the model.

The QSM model consists of a number of processors, each with private memory, that communicate via reading and writing to shared memory locations. Each synchronized phase executed by the processors consists of an arbitrary interleaving of the following operations:

- shared-memory reads: processor P_i copies values from shared memory locations into private memory.
- shared-memory writes: processor P_i writes to w_i shared memory locations.
- local computation: processor P_i performs c_i RAM computations which involve only private state and private memory.

The QSM has the following features which make it more attractive as a bridging model:

- shared-memory abstraction
- bulk-synchrony
- few parameters
- queue contention metric
- work-preserving emulation on BSP
- work-preserving emulation of BSP

The authors compare the QSM and the BSP in terms of their effectiveness as bridging models for parallel computation. The BSP consists of p processor/memory components that communicate by sending point-to-point messages. The network supporting this communication is by two parameters: g, which models bandwidth; and L, which models latency. A computation on the BSP is a series of supersteps, each of which is separated by bulk synchronization.

The authors give a work preserving emulation (an emulation is work-preserving if both models perform the same amount of work to within a constant factor) of the QSM on the BSP. However, since the QSM has fewer parameters than the BSP and does not deal directly with memory partitioning, we would expect it to be easier to design algorithms on the QSM. Designs can then be mapped onto the BSP with modest slowdown.

The (d, $\mathbf{x}$)-BSP is more detailed in its modeling of memory contention than the BSP, and is characterized by 5 parameters:

- p - number of processors
- g - bandwidth
- L - latency
- d - delay, models gap at the memory banks
- $\mathbf{x}$ - expansion, ratio of memory banks to processors

The authors also present a work-preserving emulation of the QSM on the (d, $\mathbf{x}$)-BSP when $\mathbf{x} \geq d/g$.

Additionally, a work-preserving emulation of the BSP on the QSM is presented. This emulation has a probability of failure equal to $1/n^{\delta}$, for some $\delta > 0$.

Developing a suitable model for parallel computation at a proper level of abstraction has remained a challenging problem for computer scientists. BSP and LogP have recently gained popularity as candidate bridging models. The QSM is an attractive candidate for a bridging model, in particular because it provides a simple shared memory abstraction and has a small number of parameters (p, the number of processors, and g, the bandwidth gap). Additionally, the QSM can be efficiently emulated on both the BSP and (d, $\mathbf{x}$)-BSP.

2.2 Emulations between QSM, BSP and LogP: A framework for general-purpose parallel algorithm design. [2]

PRAM is a very simple model for parallel computation that abstracts too many parameters from real machines, which turns into apparently efficient models that actually run slow on real machines. Therefore, there exists the need of more realistic models that reflect with more fidelity a machine and yet is not too complicated so that implementation is kept simple and portable. BSP and LogP are models that abstract current machines more accurately than the PRAM, but there is a simple model, QSM, which could be used instead, since it is possible to efficiently emulate it on LogP and BSP.

2.2.1 Brief description of the models

All the next models have work measured as the time-processors product.

BSP BSP (Bulk-Synchronous Parallel) takes 3 parameters:

- p processor/memory components: Components that communicate via point-to-point messages.
- Bandwidth g.
- Latency L.

A computation in this model consists of a sequence of supersteps, each separated by a bulk synchronization. In each superstep, a component can perform local computations, send and receive a set of messages. The cost

T superstep is calculated as $T = max(w, gh, L)]$ and the overall time that takes an algorithm to run is the sum of T of every superstep.

LogP LogP (Latency, overhead, gap, Processors) takes 4 parameters and a derived value:

- Processors p: Number of processors.
- Latency l: Time taken by network to transmit a message from one processor to another is at most l.
- Gap g: Least units of time taken between sending or reception of a message by a processor.
- Overhead o: Time taken by a processor to send or receive a message.
- Capacity constraint: Maximum number of messages heading to the same processor can't be larger than $\lceil l/g \rceil$

Once a processor has reached its capacity, another processor trying to send a message will stall. The time taken by a LogP algorithm is the longest time taken by any processor to end its operation.

QSM QSM (Queuing Shared Memory) consists of p processors that communicate through shared memory, but run computations on private memory. An algorithm consists of the analogous for a superstep on BSP, a synchronized phase. Concurrent reads and concurrent writes are permitted on a single synchronized phase over the same shared-memory location, but not both. Metrics:

- Processors p.
- Maximum contention k: of a synchronized phase Largest number of processors reading or writing a location.

The time cost of a phase is $max(m_{op}, gm_{rw}, k)$, where m_{op} = maximum operations performed by any processor and m_{rw} =maximum read/write operations performed by any processor and k = *maximum contention*. The time taken by an algorithm is the sum of the cost of all of its phases. s-QSM is a variation of QSM where the time cost of a phase is $max(m_{op}, gm_{rw}, gk)$

2.2.2 Emulations

The paper, in addition with results obtained in other two papers, show emulations for: LogP, s-QSM and QSM on BSP. BSP, sQSM and QSM on LogP. BSP and LogP on S-QSM. BSP, LogP and s-QSM on QSM. All of them are work-preserving emulations.

2.2.3 Algorithms

In this section, the authors present optimal algorithms for prefix sums, sample sort and list ranking on QSM. This way, they show that QSM may be considered a general-purpose model. They also present a particular cost metric for QSM algorithms: number of phases; the authors claim that the time spent on synchronization is proportional to the number of phases, and, since QSM doesn't takes directly into account latency and synchronization cost, it should be considered indirectly with this metric. All the algorithms they present have a number of phases independent to the size of the input

(See open problems).

2.2.4 Experimental results

As stated previously, emulations between all the models are work-preserving, which supports the use of QSM as a model. The estimated costs of emulating QSM are confirmed by experiments run on Armadillo, a framework for simulating multiprocessor machines, but with a discrepancy on the predicted and actual communication cost, which occurs to be irrelevant compared with the total computation cost for large problems.

3 Open Problems

If we restrict ourselves to a slowdown bounded by $polylog(n)$, there is not a known work-preserving emulation of a BSP with $g = 1$ on the EREW PRAM. However, if an emulation satisfying these properties were to be discovered, it would imply the existence of randomized algorithms with linear work and polylog time for certain problems, such as computing a random permutation. In other words, even though the EREW PRAM is often considered to be a stronger model than the BSP, the EREW PRAM may have less expressive power in some cases. [1]

Similarly, no work-preserving emulation of a CRCW PRAM on a BSP with $g = 1$ is known if we require a $polylog(n)$ slowdown. In other words, though the authors have presented a strong correspondence between the QSM and the BSP, such a strong correspondence between the BSP and PRAMs that have an additional g parameter is not known.

The main problem that parallel models are currently facing is: Is there a method that can automatically optimize the code generated for a model so that the hidden parameters become irrelevant for any algorithm? Is communication sufficiently optimizable or at least irrelevant for any algorithm given a large problem size? Could any problem be solved in few phases, referring to QSM.

A related question arise from the results of Vlr03: Would emulations correctly predict the total cost for algorithms with high level of communication? Will the emulations still be valid for machines with massive number of processors? Neural network simulation, for example, naturally benefits

from a massive number of processors and heavily relies on communication between them. Of course it is a radical example and reducing the number of phases could be very challenging, if not impossible, but it shows that a method to analyze the nature of a problem is needed in order to know if QSM is applicable.

4 Emulation of BSP on LogP

5 Emulation of LogP on BSP

Theorem 1: There is a deterministic work-preserving emulation of a p-processor non-stalling LogP on BSP with slowdown $O(L/l)$. (Theorem 3.7 in [2])

Theorem 2: There is no deterministic work-preserving step-by-step emulation of a p-processor stalling LogP on BSP. (Theorem 3.9*b* in [2])

Proof of Theorem 1: The emulation is as follows:

- Map the LogP processors evenly among the BSP processors, so that each BSP processor is assigned L/l LogP processors.
- Divide the LogP computation into blocks of computation of length l.
- Let each BSP processor emulate the steps performed by each LogP processor in the current block of computation.

Since each BSP processor is assigned L/l LogP processors, the BSP has a total of $(l/L)P$ processors. Note that the emulation of each block will take 2 BSP supersteps. Each block of LogP computation is of length l, and each message takes exactly l units of time to each its destination, so messages sent within block i will arrive at their destinations within block $i+1$. In the BSP emulation, each BSP processor adds an additional tag to each message; this tag stores the LogP step in which the message was sent. A BSP processor examines the messages it received from the previous block, sorts them by tags in $O(L)$ time, using counting sort, and processes the received messages

in sorted order. This means that the emulation executes each of the LogP steps in the same order as the original LogP execution.

The analysis of the time cost of emulating one block is as follows: Recall that the time cost of a single BSP superstep is $w + gh + L$, where w is the maximum number of local operations performed by any processor and h is the maximum number of messages sent or received by any processor. In this case $w \leq (L/l)l = L$, since each BSP processor is responsible for emulating the local computations of L/l LogP processors across a block of length l. The number of messages sent by a BSP processor is at most L/g, because each of the emulated L/l processors can send at most l/g messages during a block of length l. Similarly, since the LogP computation under consideration is non-stalling, the number of messages received by a BSP processor is at most L/g. Thus, $h \leq L/g$, from which it follows that the time cost of emulating one LogP block is $O(L)$. However, the time cost of one block under the original LogP execution is $O(l)$, which means that the slowdown of the emulation is $O(L/l)$.

The work cost of one block under the original LogP execution is $O(pl)$. The work cost of one block under the BSP emulation is $O((l/L)p \cdot L) = O(pl)$. Hence, the emulation is work-preserving. This completes the proof of Theorem 1.

Proof of Theorem 2: Consider the following computation, to be run on a LogP with $p = r \cdot (q + 1)$ processors, grouped into r groups of q processors and one group of r processors ($p_{i,j}$ for $1 \leq i \leq r$ denotes the jth processor in group i; and P_i denotes the ith processor in the group with r processors):

pfor $1 \leq j \leq r$

$p_{1,j}$ sends a message to processor $p_{2,j}$

$p_{1,j}$ sends a message to processor P_1

end

pfor $2 \leq i \leq r$

pfor $1 \leq j \leq q$

if processor $p_{i,j}$ receives a message from processor $p_{(i-1),j}$ **then**

if $i < r$:

processor $p_{i,j}$ sends a message to processor $p_{(i+1),j}$

end

processor $p_{i,j}$ sends a message to processor P_i

end

end

end

Claim: The above computation takes time $O(r \cdot l + g \cdot q)$ on the LogP, but when emulated on the BSP takes time $\Omega(r \cdot (L + g \cdot q))$.

Proof: At time $(i-1)l + g$, all processors in the ith group send a message to processor P_i (note that when q, the size of each of the r groups, exceeds l/g, this is a stalling LogP computation). Processor P_i, however, will not receive these messages until time $i \cdot l + g \cdot q$. Then, the computation will terminate at time $r \cdot l + g \cdot q$, when P_r receives all messages.

However, on the BSP, this computation must be executed in r supersteps. On the BSP, any send based on the condition of receiving a message in the current superstep cannot be executed in the same superstep. This means that

the processors in group i, for $2 \leq i < r$ can send messages only after having received a message from a processor in group $i-1$. Furthermore, note that in each superstep, q messages are received by some processor (P_i in superstep i). Thus, the emulated computation takes time $\Omega(r \cdot (L + g \cdot q)) = \Omega(r \cdot L + r \cdot g \cdot q)$. This means that the slowdown of the emulation in this case is $\Omega(\frac{r \cdot L + r \cdot g \cdot q}{r \cdot l + g \cdot q})$ It is interesting to note that since there is no local computation in this algorithm, the parameter o does not appear anywhere in the analysis.

Lemma: Any deterministic step-by-step emulation of LogP on BSP can have arbitrarily large slowdown. (Proof of this is given in the proof of Theorem 3.9 in [2])

Now, suppose we have a work-preserving emulation e of stalling LogP on BSP. Let the slowdown of e be τ. Now consider e applied to the above algorithm. We let $r = \omega(\tau)$, $r \cdot l = o(g \cdot q)$. Furthermore, assume $l \leq L$, which is not unreasonable because l accounts for latency while L accounts for latency and synchronization. The work done by the LogP computation is $\Theta(g \cdot q \cdot p)$. In this case, the emulation takes time $\Omega(r \cdot g \cdot q)$. Recall that the BSP has $1/\tau \cdot p$ processors, so the work of the emulated algorithm is $\Theta(r \cdot g \cdot q \cdot \frac{p}{\tau}) = \omega(g \cdot q \cdot p)$. But this means that e is not work-preserving, so there can be no work-preserving emulation of stalling LogP on BSP.

References

[1] P. Gibbons, Y. Matias, V. Ramachandran. "Can a shared memory model serve as a bridging model for parallel computation?" Theory of Computing Systems Special Issue on papers from SPAA'97, vol. 32, no. 3, 1999, pp. 327-359

[2] V. Ramachandran, B.Grayson, M. Dahlin.(2003) "Emulations between QSM, BSP and LogP: A framework for general-purpose parallel algorithm design." Journal of Parallel and Distributed Computing, vol. 63, 2003, pp. 1175-1192.

[3] L. G. Valiant. "A Bridging Model for Parallel Computation." (1990) Communications of the ACM, 33(8) pp. 103-111

[4] V. Ramachandran. "A General Purpose Shared Memory Model for Parallel Computation" Algorithms for Parallel Processing, vol. 105. IMA Volumes in Mathematics and Its Applications

[5] P. B. Gibbons. "What Good are Shared Memory Models?" (1996) ICPP Workshop on Challenges for Parallel Processing