ALGORÍTMOS PARALELOS
(Aula 4)

Neyval C. Reis Jr.

OUTUBRO/2004

Laboratório de Computação
de Alto Desempenho

DI/UFES

Programa do Curso

1. Introdução
2. Arquitetura de Computadores
3. Arquiteturas de Sistemas Paralelos
4. Computação de Alto Desempenho
5. Programação Paralela (modelos e paradigmas) ⟵
6. Análise de Desempenho e Instrumentação
7. Aplicações

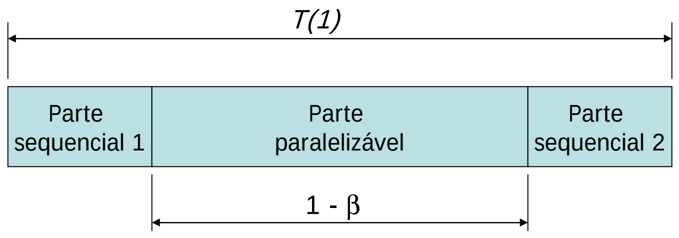
Programa do Curso

5. Programação Paralela (modelos e paradigmas)
   a) Começando a pensar em paralelo (exemplo)
   b) Metodologia de design
   c) Paradigmas de Programação
   d) Eficiência
   e) Ferramentas

Fundamentos de metodologia de design

Lei de Amdahl (1967)

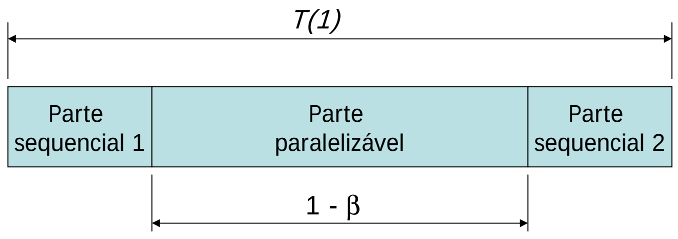
Tempo de execução em um único processador:



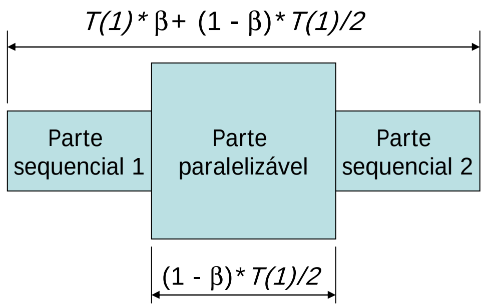
β = fraçao do programa que é sequencial

Lei de Amdahl

Tempo de execução em um único processador:



Tempo de execução em 2 processadores:

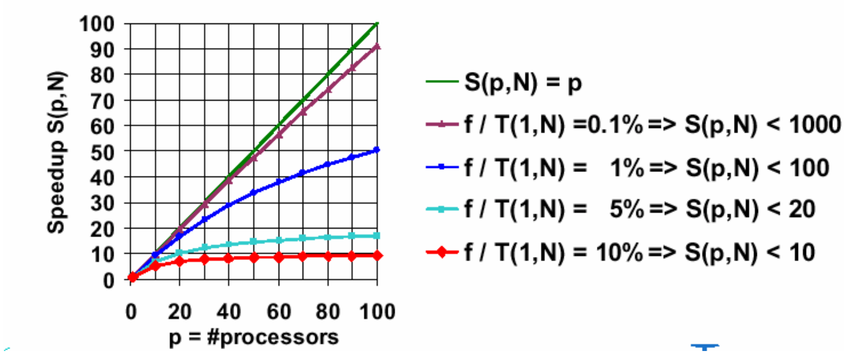


Lei de Amdahl

T(1,N) = f + (T(1,N) - f)   f ... sequential part of code that can not be done in parallel

S(p,N) = T(1,N) / T(p,N) = T(1,N) / (f + (T(1,N) - f) / p)

For p —> infinity, speedup is limited by S(p,N) < T(1,N) / f



Lei de Gustafson-Barsis

• A lei de Amdahl tornou-se um incômodo para os fabricantes de máquinas de grande porte.

• No final da década de 80, Gordon Bell ofereceu um prêmio anual de US$ 1000,00 para quem pudesse utilizar processamento paralelo de maneira eficiênte para resolver problemas reais (lista de ganhadores).

• Em 1987, um grupo de pesquisadores do Laboratório de Sandia (Laboratório Computaçao Científica - EUA) obtiveram speed-ups de aproximadamente 1000 para problemas com β entre 0,004 e 0,008, eqnquanto que a lei de Amdahl previa speed-ups de apenas 125 a 250 para estes casos.

Lei de Gustafson-Barsis

• John Gustafson e Ed Barsis formalizaram o conceito por básico da aparente contradiçao. A chave é que Amdahl assume que o valor de β seja constante para qualquer valor de *p* (n. processadores), de fato (1- β) quase nunca é independente de N.

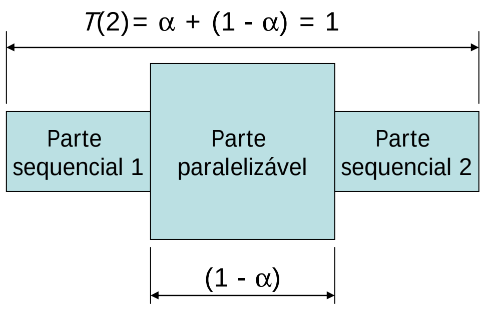
• Lei de Gustafson-Barsis

$$S = T(1)/T(p)$$
$$T(p) = 1$$
$$T(p) = \alpha + (1-\alpha) = 1$$

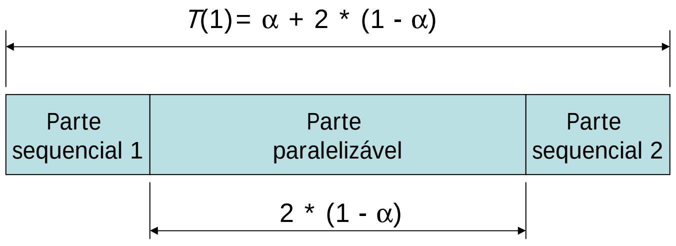
fraçao sequencial do programa

Lei de Gustafson-Barsis

Tempo de execução em 2 processadores:



Tempo de execução em um único processador:



Lei de Amdahl

$$S = \frac{1}{\left(\beta + \frac{(1-\beta)}{p}\right)}$$

S **=Speed-up**

p **= n. de porcessadores**

β = fraçao sequencial do programa

Lei de Gustafson-Barsis

$$S = p - \alpha(p-1)$$

S **=Speed-up**

p **= n. de porcessadores**

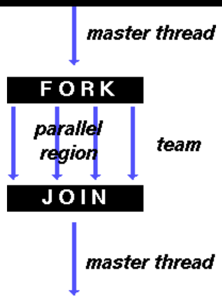
α = fraçao sequencial do programa

Metodologia de design

Fork-Join

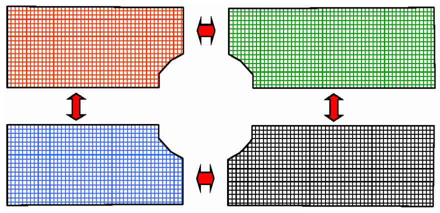
Paradigma Fork-Join (decomposiçao

de tarefas e junçao dos resultados)



data-parallel

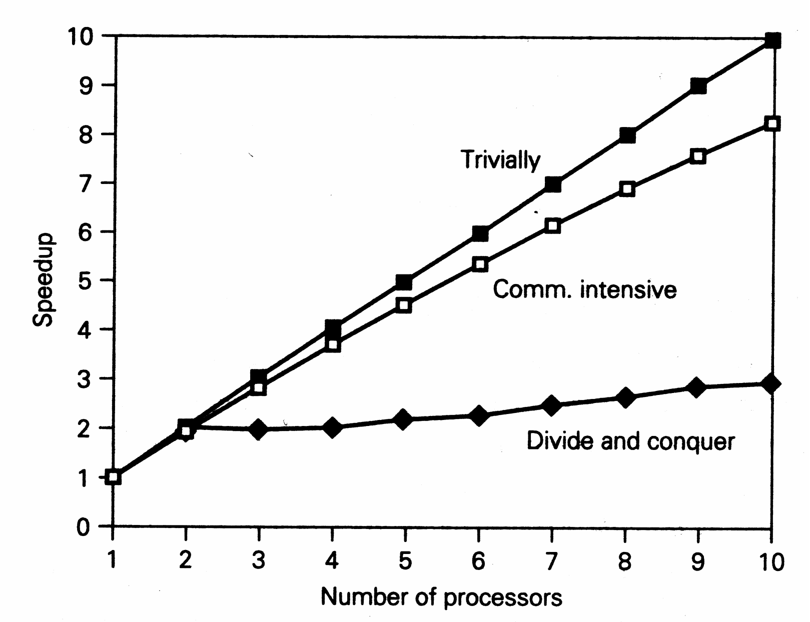
Paradigma data-parallel
(decomposiçao de tarefas
devida a decomposiçao
dos dados)



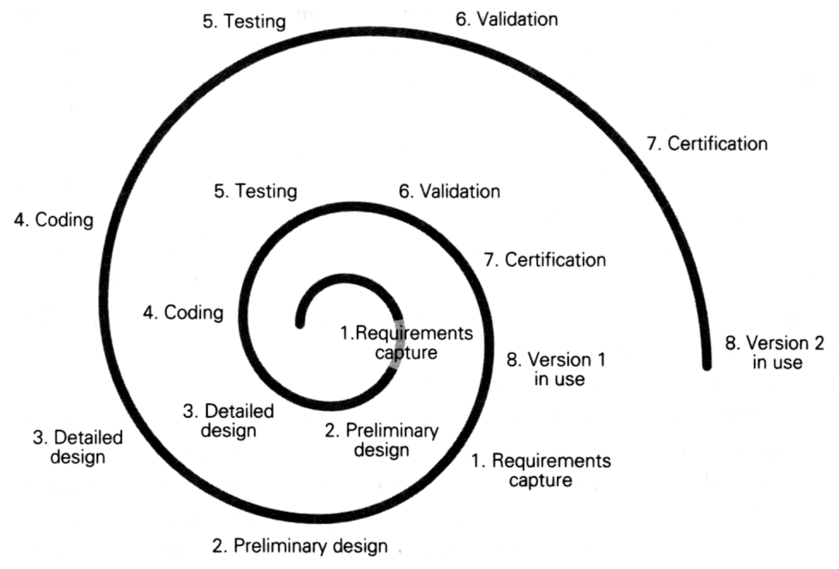
Limites para o paralelismo

- Trivialmente paralelo (trivially parallel) → S = p
    - exemplo: aplicaçoes “bag of tasks”
- Dividir e conquistar (Divide and Conquer) → $S = p / \log_2 p$
    - exemplo: aplicaçoes baseadas em árvores binária, como busca, ordenaçao, somatórias, etc.
- Paralelismo limitado por comunicaçao (Communication bound Paralelism) → $S = 1 / C(p)$
    - *em geral $C(p) = A+B(p)$,* onde A e B dependem da forma de comunicaçao entre processadores e memória (latência e bandwidth)
    - exemplo: multiplicaçao de matrizes e métodos numéricos
- Super-linear paralelismo → S > p
    - exemplo: uso mais eficiente da hierarquia de memória

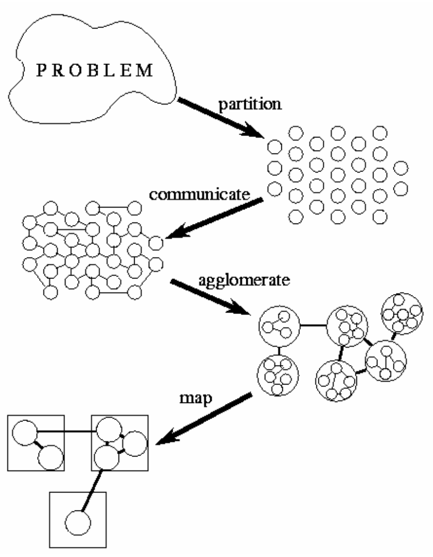
Limites para o paralelismo



Desenvolvimento



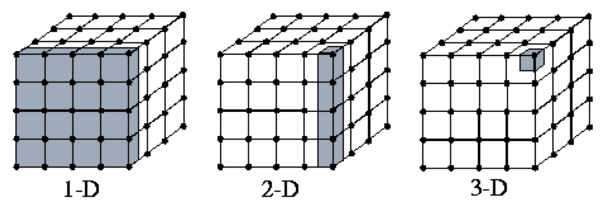
Desenvolvimento



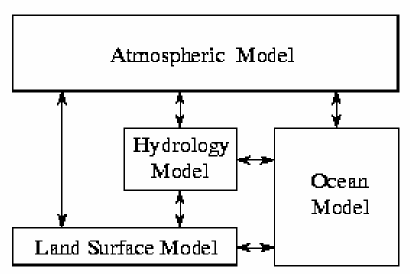
1.Partitioning. The computation that is to be performed and the data operated on by this computation are decomposed into small tasks. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution.
2.Communication. The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.
3.Agglomeration. The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.
4.Mapping. Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

Particionamento

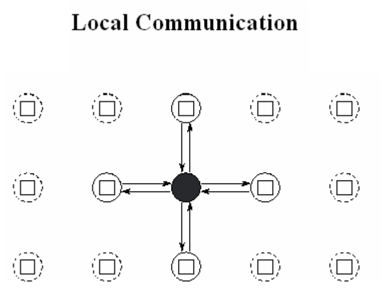
Dados:

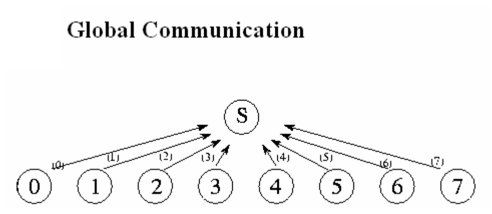


Funcional:



Comunicaçao





Comunicaçao

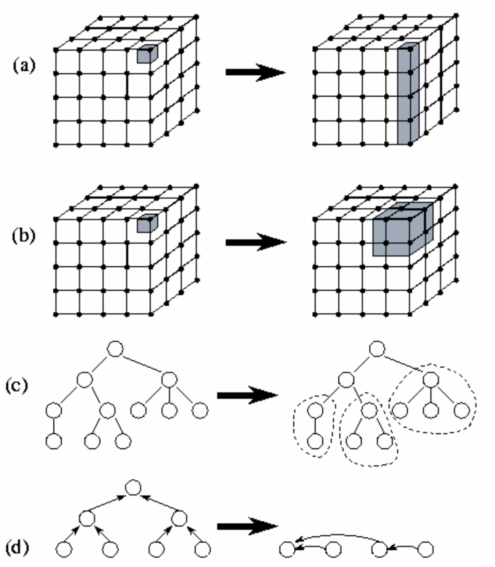
Having devised a partition and a communication structure for our parallel algorithm, we now evaluate our design using the following design checklist. However, we should be aware of when a design violates them and why.

1.Do all tasks perform about the same number of communication operations?

2.Does each task communicate only with a small number of neighbors?

3.Are communication operations able to proceed concurrently? Is the computation associated with different tasks able to proceed concurrently? If not, your algorithm is likely to be inefficient and nonscalable. Consider whether you can reorder communication and computation operations.

Aglomeraçao



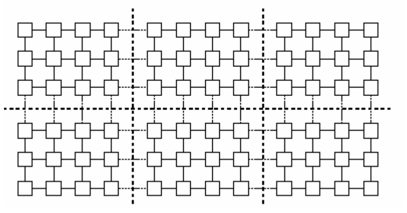
Aglomeraçao

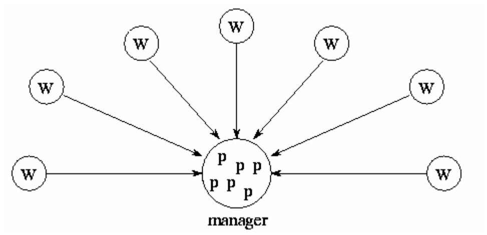
1.Has agglomeration reduced communication costs by increasing locality?
2.If agglomeration has replicated computation, have you verified that the benefits of this replication outweigh its costs, for a range of problem sizes and processor counts?
3.If agglomeration replicates data, have you verified that this does not compromise the scalability of your algorithm by restricting the range of problem sizes or processor counts that it can address?
4.Has agglomeration yielded tasks with similar computation and communication costs? The larger the tasks created by agglomeration, the more important it is that they have similar costs. If we have created just one task per processor, then these tasks should have nearly identical costs.
5.Does the number of tasks still scale with problem size?
6.If agglomeration eliminated opportunities for concurrent execution, have you verified that there is sufficient concurrency for current and future target computers?
7.Can the number of tasks be reduced still further, without introducing load imbalances, increasing software engineering costs, or reducing scalability?
8.If you are parallelizing an existing sequential program, have you considered the cost of the modifications required to the sequential code?

Divisao de tarefas (Mapping)

Our goal in developing mapping algorithms is normally to minimize total execution time. We use two strategies to achieve this goal:

1. We place tasks that are able to execute concurrently on *different* processors, so as to enhance concurrency.
2. We place tasks that communicate frequently on the *same* processor, so as to increase locality.





Modelos de Programação

Porque precisamos de modelos ?

• São os modelos formais que nos ajudam a demonstrar que os algoritmos são ótimos ou a obter "ótimos" resultados.

• Sua importância também está relacionada a possibilidade de relacionar a complexidade do algoritmo sequencial com o algoritmo paralelo.

• Através da remoção dos detalhes de codigicação, como comunicação e sincronição, o programador pode focar-se nas caracteristicas estruturais do problema e do processamento concorrente.

Análise de algoritmos paralelos

De uma maneira geral a performance de um algoritmo paralelo é medida com base no tempo de execução e na quantidade de recursos consumidos:

- **Tempo de Execução:** a quantidade de tempo total gasta na execução do programa (**T**)
- **Número de processadores** necessários para a execução do algoritmo (**P**)
- **Custo** do algoritmo é definido como o produto entre o número de processadores e o tempo de execução (**C**).

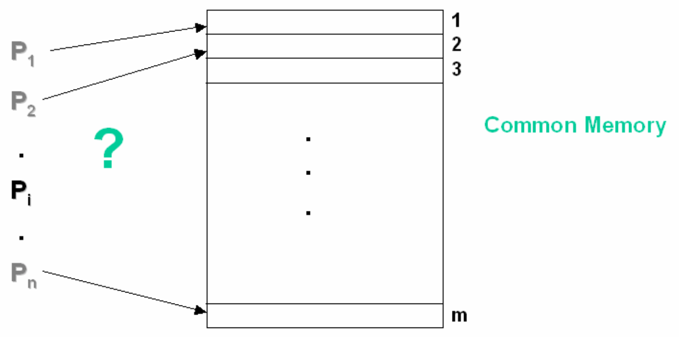
Limites dos Modelos:

- Sistemas com arquiteturas diferentes precisam de algoritmos diferentes para resolver o mesmo problema.
- A análise e descrição de problemas reais é geralmente bastante difícil, uma vez que qualquer acesso de memória ou migração de dados devem ser descritos.

PRAM

- Parallel Random Access Machine
- Shared-memory multiprocessor
- unlimited number of processors, each
  - has unlimited local memory
  - knows its ID
  - able to access the shared memory
- unlimited shared memory

PRAM



PRAM n RAM processors connected to a common memory of m cells

ASSUMPTION: at each time unit each P_i can read a memory cell, make an internal computation and write another memory cell.

CONSEQUENCE: any pair of processor P_i P_j can communicate in constant time!

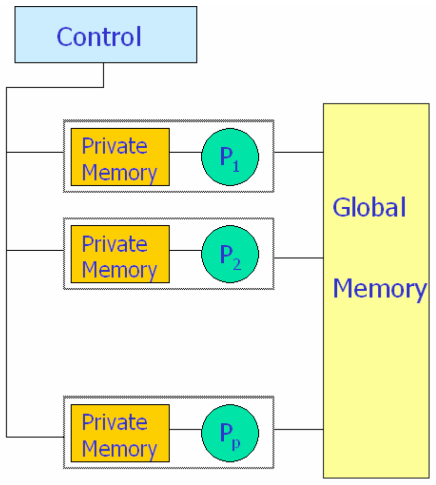
P_i writes the message in cell x at time t
P_j reads the message in cell x at time t+1

PRAM

- Inputs/Outputs are placed in the shared memory (designated address)
- Memory cell stores an arbitrarily large integer
- Each instruction takes unit time
- Instructions are synchronized across the processors

PRAM

- Synchronized Read Compute Write Cycle
- EREW
- ERCW
- CREW
- CRCW
- Complexity:
  T(n), P(n), C(n)



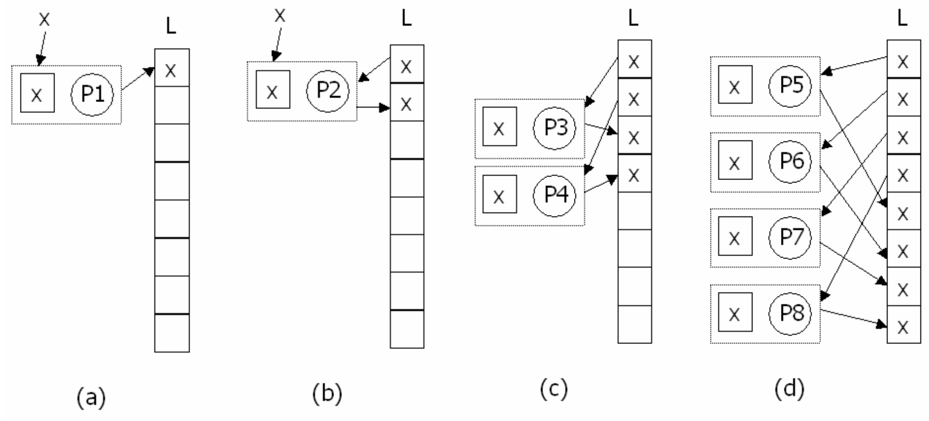
Modelo PRAM e suas variações

- There are different modes for read and write operations in a PRAM.
  - Exclusive read(ER)
  - Exclusive write(EW)
  - Concurrent read(CR)
  - Concurrent write(CW)
    - Common
    - Arbitrary
    - Minimum
    - Priority
- Based on the different modes described above, the PRAM can be further divided into the following four subclasses.
  - EREW-PRAM model
  - CREW-PRAM model
  - ERCW-PRAM model
  - CRCW-PRAM model

Simulação de um acesso múltiplo a um dado, utilizando o modelo PRAM EREW

- Broadcasting mechanism:
  - P1 reads x and makes it known to P2.
  - P1 and P2 make x known to P3 and P4, respectively, in parallel.
  - P1, P2, P3 and P4 make x known to P5, P6, P7 and P8, respectively, in parallel.
  - These eight processors will make x know to another eight processors, and so on.

Simulação de um acesso múltiplo a um dado, utilizando o modelo PRAM EREW



Simulating Concurrent read on EREW PRAM with eight processors using Algorithm Broadcast_EREW

Simulação de um acesso múltiplo a um dado, utilizando o modelo PRAM EREW

- **Algorithm Broadcast_EREW**

```
Processor $P_1$
        y (in $P_1$'s private memory) $\leftarrow$ x
        L[1] $\leftarrow$ y
for i=0 to log p-1 do
        forall $P_j$, where $2^i + 1 \leq j \leq 2^{i+1}$ do in parallel
                y (in $P_j$'s private memory) $\leftarrow$ L[j-$2^i$]
                L[j] $\leftarrow$ y
        endfor
endfor
```

THE PRAM IS A THEORETICAL (UNFEASIBLE) MODEL

• The interconnection network between processors and memory would require a very large amount of area .

• The message-routing on the interconnection network would require time proportional to network size (i. e. the assumption of a constant access time to the memory is not realistic).

WHY THE PRAM IS A REFERENCE MODEL?

• Algorithm's designers can forget the communication problems and focus their attention on the parallel computation only.

PRAM

One should notice, however, that PRAM algorithms, when implemented in practice, leave much to be desired in terms of actual performance. Frequently speedup results for theoretical PRAM algorithms do not match the actual speedups obtained in experiments performed on real parallel computers. So in spite of the usefulness, as far theory is concerned, of the PRAM model, we are desperately in need of more realistic parallel computing models.

BSP - Bulk Synchronous Parallel

Among the realistic computing models, the most important is probably Valiant's BSP (Bulk Synchronous Parallel) computing model, proposed in 1990. A BSP computer consists of a set of processor/memory modules connected by a router that can deliver messages in a point to point fashion among the processors. In the BSP model, computation is divided into a sequence of supersteps separated by barrier synchronizations. A superstep in turn consists of local computation and data exchange among processors through the router. Though BSP is possible to simulate PRAM algorithm optimally on distributed memory machines, Valiant observes the importance of design of parallel algorithms that take advantage of local computations and minimize global operations. Valiant also points out situations in which PRAM simulations are not efficient and these situations, unfortunately, occur in the majority of current parallel computers.

BSP - Bulk Synchronous Parallel

Bulk synchronous parallelism (BSP)is a model in which interconnection network properties are captured by a few architectural parameters. A BSP abstract machine consists of a collection of ***p*** abstract processors, each with local memory, connected by an interconnection network whose only properties of interest are the time to do a barrier synchronization (***l***) and the rate at which continuous randomly addressed data can be delivered (***g***). These BSP parameters are determined experimentally for each parallel computer.

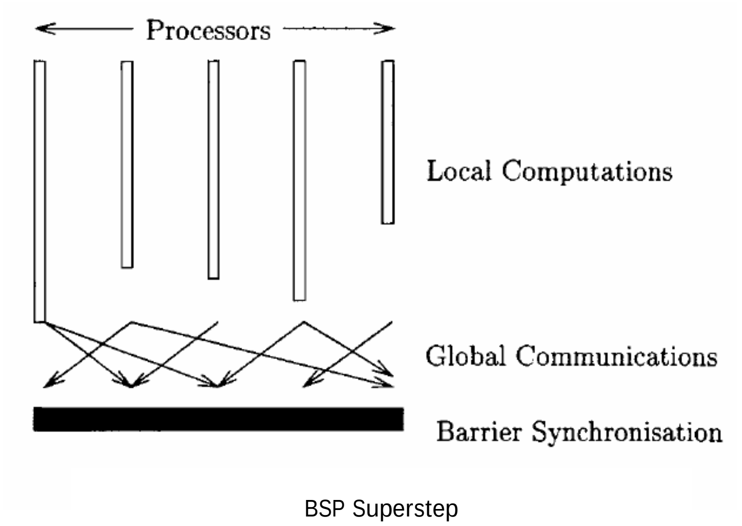
BSP - Bulk Synchronous Parallel

A BSP (abstract) program consists of *p* threads and is divided into *supersteps.* Each superstep consists of: a computation in each processor, using only locally held values; a global message transmission from each processor to any set of the others; and a barrier synchronization. At the end of a superstep, the results of global communications become visible in each processor's local environment. If the maximum local computation on a step takes time *w* and the maximum number of values sent by or received by any processor is h, then the total time for a superstep is given by

$t = w + hg + l$

(where g and l are the network parameters), so that it is easy to determine the cost of a program. This time bound depends on randomizing the placement of threads and using randomized or adaptive routing to bound communication time.

BSP - Bulk Synchronous Parallel



BSP Superstep

BSP - Bulk Synchronous Parallel

Thus BSP programs must be decomposed into threads, but the placement of threads is then done automatically. Communication is implied by the placement of threads, and synchronization takes place across the whole program. The model is simple and fairly abstract, but lacks a software construction methodology.

The cost measures give the real cost of a program on any architecture for which *g* and *l* are known.

The current implementation of BSP uses an SPMD library that can be called from C and FORTRAN. The library provides operations to put data into the local memory of a remote process, to get data from a remote process, and to synchronize.

LogP

Another related approach is LogP [Culler et al. 1993], which uses similar threads with local contexts, updated by global communications. However, LogP does not have an overall barrier synchronization. The LogP model is intended as an abstract model that can capture the technological reality of parallel computation. LogP models parallel computations using four parameters:

L: an upper bound on the *latenc*y, or delay, incurred in communicating a message containing a word (or small number of words) from its source module to its target module.
o: the *overhea*d, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.
g: the *ga*p, defined as the minimumtime interval between consecutive message transmissions or consecutive message receptions at a processor.
P: the number of processor/memory modules. We assume unit time for local operations and call it a cycle.

LogP

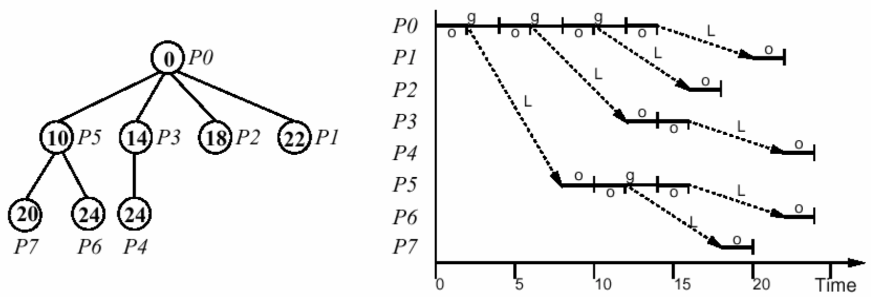


Figure 3: *Optimal broadcast tree for $P = 8, L = 6, g = 4, o = 2$ (left) and the activity of each processor over time (right). The number shown for each node is the time at which it has received the datum and can begin sending it on. The last value is received at time 24.*

LogP

A set of programming examples has been designed with the LogP model and implemented on the CM-5 parallel machine to evaluate the model's usefulness. However, the LogP model is no more powerful than BSP [Bilardi et al. 1996], so BSP's simpler style is perhaps to be preferred.

Bibliografia Recomendada:

Modelos

- Models and Languages for Parallel Computation, D*avid B. Skilicorn and Domenico Talia*, disponível em http://www.inf.ufes.br/~raulh/ufes/teaching/courses/pp/page/texts/p123-skillicorn.pdf

- Designing and Building Parallel Programs, by *Ian Foster*, disponível em http://www-unix.mcs.anl.gov/dbpp/

PRAM

- Parallel Random Access Machine (PRAM) Model Emulator, Yueh-Lin Liu and Siripong Kaewyou, disponível em http://cs-www.bu.edu/faculty/best/crs/cs551/projects/pram/pram.html

Bibliografia Recomendada:

BSP (disponíveis em http://www.scs.carleton.ca/~bsp/):

• Silvia Goetz, "Algorithms in CGM, BSP and BSP* Model: A Survey", Term Paper for Graduate Course "Parallel Algorithms and VLSI Implementation", Carleton Unviversity, Ottawa, January 1997. **on Parallel Computation Models**

• B.M. Maggs, L.R. Matheson, R.E. Tarjan, "Models of Parallel Computation: A Survey and Synthesis, *Proc. of the 28th Hawaii International Conference on System Sciences (HICSS)*, Vol. 2, Jan. 1995, pp. 61-70. S. E. Hambrusch, "Models for Parallel Computation", *Proceedings of Workshop on Challenges for Parallel Processing*, International Conference on Parallel Processing, 1996.

LogP (disponíveis em http://www.scs.carleton.ca/~bsp/other.html#Log):

• D.E.Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, T. von Eicken, "LogP:Towards a Realistic Model of Parallel Computation", *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

• A. Alexandrov, M. Ionescu, K. E. Schauser, C. Scheiman, "LogGP: Incorporating Long Messages into the LogP model - One step closer towards a realistic model for parallel computation", *7th Annual Symposium on Parallel Algorithms and Architecture (SPAA'95)*, July 1995.