Anais do
III Seminário Eniac 2011
III Encontro Da Engenharia Do Conhecimento Eniac
III Encontro De Iniciação Científica Eniac

UMA APLICAÇÃO DE PROCESSAMENTO PARALELO PARA SISTEMA EM TEMPO REAL

Cao Ji Kan[1]
Jefferson Perez R. Costa[2]

1 Doutor em Engenharia Elétrica pela EPUSP e Bel. em Ciência da Computação, Professor no Curso de Sistemas de Informação na Faculdade de Tecnologia ENIAC.
2 Mestre em Engenharia Elétrica pela EPUSP, e Bel. em Ciência da Computação, Professor no Curso de Sistemas de Informação na Faculdade de Tecnologia ENIAC.

1. INTRODUÇÃO

Sistemas de processadores paralelos oferecem um potencial enorme para a redução do tempo de execução de aplicações que requerem cálculos intensivos. A execução de programas em tais sistemas paralelos, se devidamente paralelizados, pode ser acelerada de modo substancial em relação à execução sequencial.

Praticamente todos os sistemas de tempo real são inerentemente concorrentes. Isso porque um dos objetivos mais importantes no sistema de tempo real é que o sistema deve corresponder ao domínio do problema (mundo real) na escala de tempo deste mundo real. Com este objetivo, podemos usar várias técnicas de paralelização para paralelizar os processos sequenciais autônomos no sistema para atingir o alto desempenho.

Neste trabalho, mostramos um exemplo de uma aplicação de processamento paralelo para o sistema de tempo real e urna implementação para o algoritmo paralelo utilizando a técnica denominada de pipeline. Neste trabalho vamos adotar o modelo de computadores paralelos de memória distribuída com paradigma *SPMD* (*Single Program Multiple Data*). A arquitetura deste modelo consiste de multiprocessadores independentes interligados através de uma memória compartilhada. Cada processador tem sua memória própria ou local (ver figura 1).

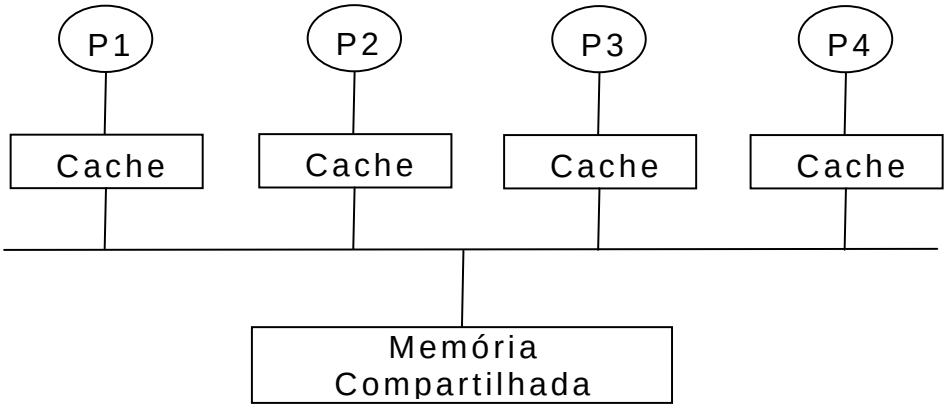


Figura 1: Arquitetura para multiprocessadores com a memória compartilhada

SPMD é um modelo de operação assíncrona que significa "rodando o mesmo programa com dados diferentes". Pois um programa completo executado em dados separados pode causar desvios diferentes, conduzindo, assim, ao paralelismo assíncrono.

Apesar de executarem o mesmo programa, os processadores não estão fazendo exatamente a mesma coisa em cada passo, ou seja, eles estão executando instruções diferentes de mesmo programa.

2 SIMULAÇÃO DE CONTROLE DE MANIPULADORES ATRAVÉS DE PROCESSAMENTO PARALELO

2.1 Controle de Manipuladores

Os sistemas de controle de manipuladores apresentam diversos requisitos relacionados ao desempenho. Dentre estes requisitos está o tempo de resposta do sistema. A movimentação realizada por um manipulador é geralmente estipulada previamente através de urna determinada trajetória. Assim, o percurso a ser realizado pelo manipulador deverá obedecer uma certa orientação espacial.

A posição atual é fornecida através de sensores, que informam ao sistema de controle a referência para as subsequentes movimentações dos atuadores do robô. A figura 2 ilustra este ciclo executado pelos sistemas em robótica.

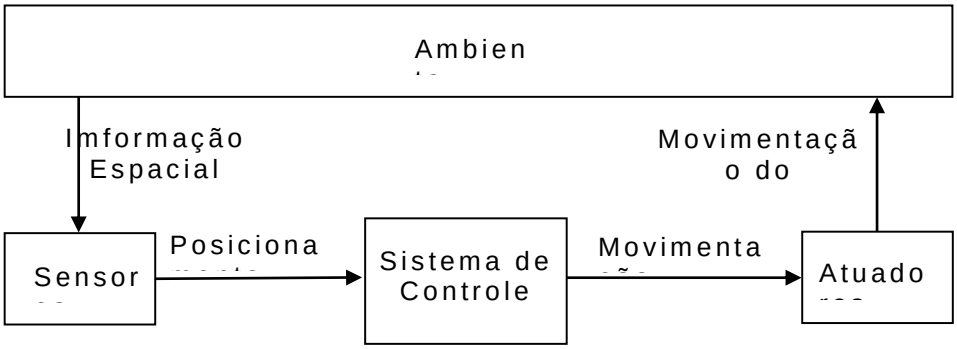


Figura 2: Sistema de robótica

O ciclo do sistema se inicia no recebimento das informações pelos sensores, posteriormente passa pelo sistema de controle digital e termina através de uma resposta dada pelos atuadores.

Por motivos de desempenho, todo este ciclo não deve demorar que 10 ms. Assim, os sensores devem apresentar uma rápida resposta e a movimentação dos atuadores deve ser imediata. Além disso, o sistema de controle deve realizar os cálculos no menor tempo possível. Para se alcançar um alto desempenho nos cálculos realizados pelo controle, diversos algoritmos paralelos podem ser aplicados. A seguir será descrita uma solução utilizando a técnica denominada de *pipeline*.

Os sistemas de robótica utilizados atualmente são fragmentados em três partes principais:

Sensores;
Sistema de Controle;
Atuadores;

Os sensores são dispositivos responsáveis pela captura de informações do robô em relação ao ambiente. Tais informações servem de referencia para o sistema de controle, que é responsável pelos cálculos de movimentação dos atuadores.

Os atuadores são responsáveis por realizar movimentações com a finalidade de se cumprir uma determinada atividade. A figura 3 ilustra um manipulador com duas articulações.

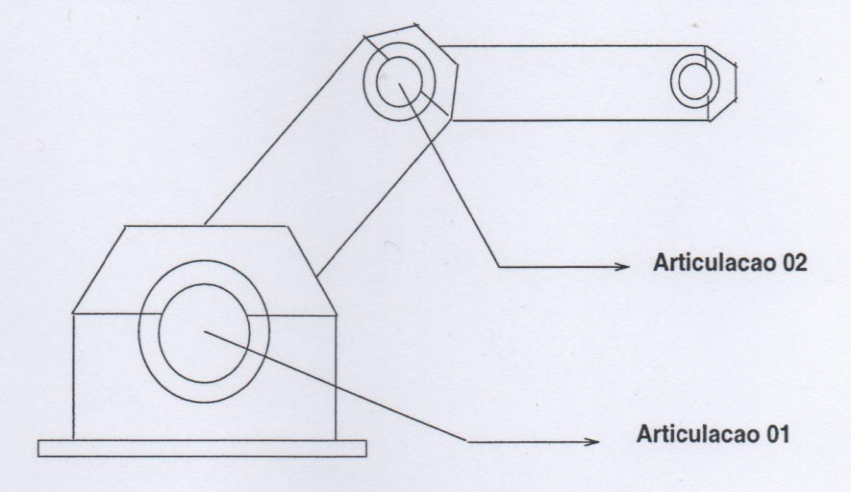


Figura 3: Manipulador com Duas Articulações

A trajetória que o manipulador deve percorrer no espaço é especificada através de coordenadas cartesianas. Entretanto, os atuadores geralmente operam através do deslocamento ou da velocidade angular.

Desta forma, uma convensão de coordenadas se faz necessária, sendo que este cálculo deve ser realizado no menor tempo possível para que sistema não seja onerado.

Basicamente, a conversão pode ser realizada através das seguintes equações:

$$\cos(\theta_2) = \frac{x^2 + y^2 - l_1^2 - l_2^2}{2 \cdot l_1 \cdot l_2}$$

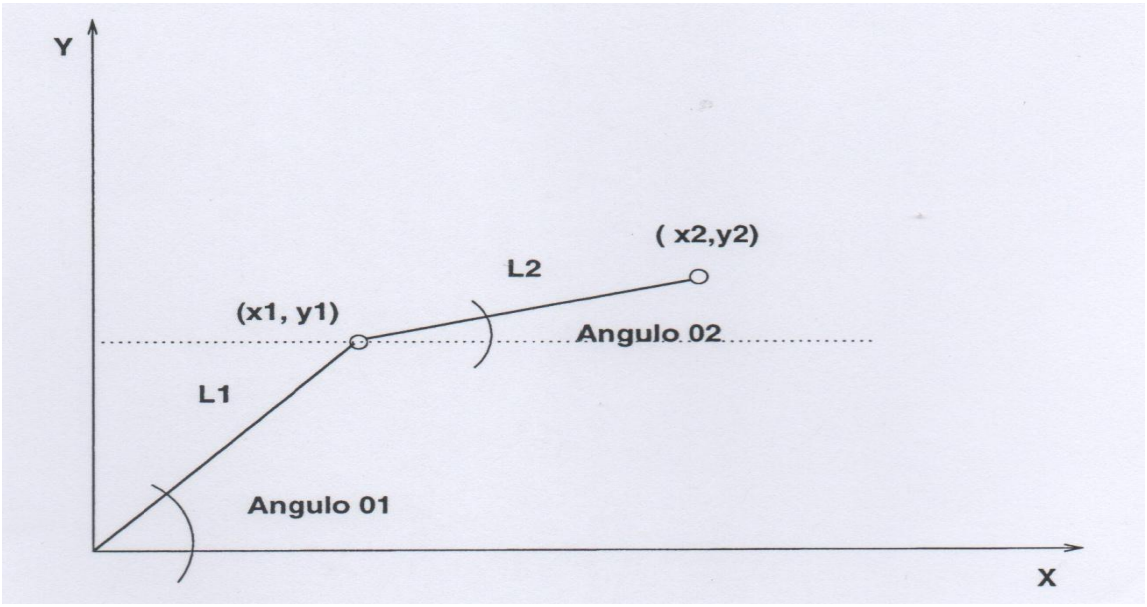
Esta equação pode ser rescrita da seguinte forma.:

$$\theta_2 = \arccos\left(\frac{x^2 + y^2 - l_1^2 - l_2^2}{2 \cdot l_1 \cdot l_2}\right)$$

$$\theta_1 = \arccos\left(\frac{x}{y}\right) - \arctan\left(\frac{l_2 \cdot \sin\theta_2}{l_1 + l_2 \cdot \cos\theta_2}\right)$$

Sendo que l_1 e l_2 correspondem ao comprimento de cada parte do braço do manipulador. Nas equações anteriores, as coordenadas cartesianas *x* e *y* são transformadas nos ângulos θ_1 e θ_2. A figura 4 mostra as coordenadas cartesianas *x* e *y*.

Figura 4: As Coordenadas Cartesianas x e y



2.2 Algoritmo de Pipeline

Primeiro vou mostrar a ideia de *pipeline*. A execução de uma determinada tarefa pode ser desdobrada em várias estágios, cada um executado numa unidade independente (como uma linha de montagem, por exemplo, urna linha de montagem de carro.). Se cada unidade leva *n* ns, uma tarefa leva 5*n* ns para ser executada. Porém se todas as unidades podem ser mantidas sempre ocupadas, então é possível completar essa tarefa em cada *n* ns.

Com base nessa ideia, o algoritmo de *pipeline* consiste na divisão de uma determinada tarefa em fases que podem ser executadas paralelamente utilizando dados distintos. Assim, se urna tarefa pode ser fragmentada em quatro fases, então é possível se executar quatro processos paralelamente.

A ilustração seguinte exibe quatro processadores que executam fases distintas de uma determinada tarefa.

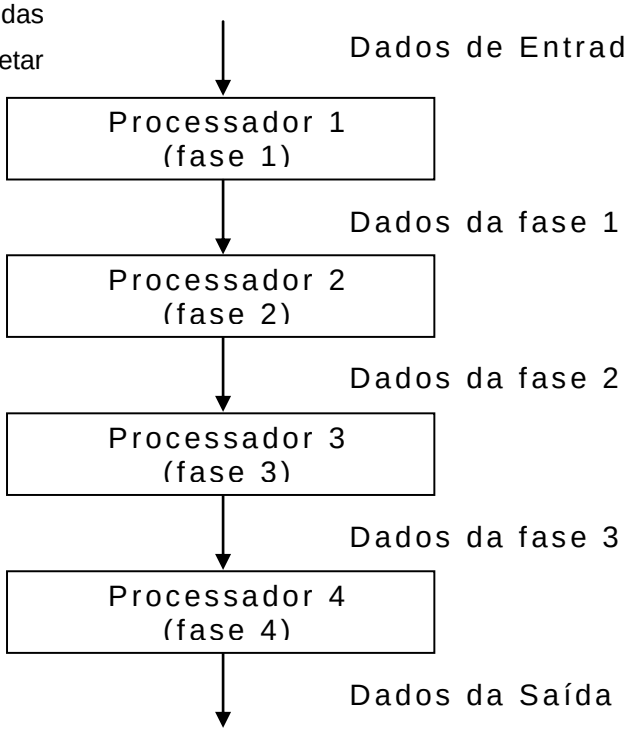
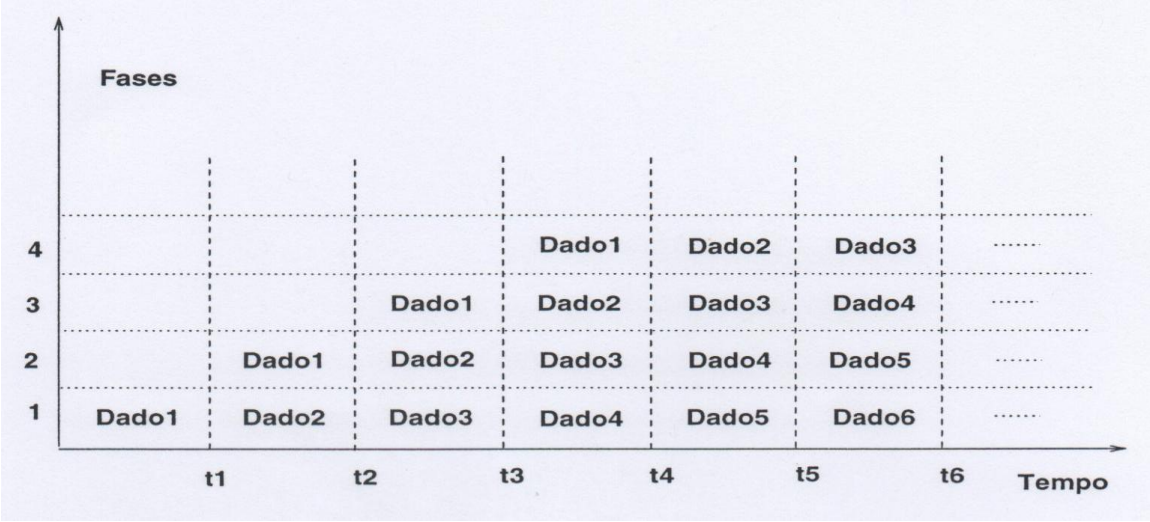


Figura 5: Ilustração de *Pipeline*

No instante que o processador P1 calcula os dados da fase 1, os demais processadores estarão calculando os dados das demais fases. O gráfico seguinte exibe a disposição das fases para o cálculo de diversos dados no *pipeline*. Dentre as vantagens deste algoritmo está a execução paralela das fases da tarefa, diminuindo drasticamente o tempo de finalização do cálculo dos dados. A figura 5 mostra a distribuição de dados no *pipeline*. O tempo gasto em cada dado é praticamente o mesmo que no processo sequencial, entretanto, no sistema *pipeline*, as fases da tarefa são liberados em um tempo bem mais reduzido. Considerando a transformação das coordenadas cartesianas em coordenadas polares como urna tarefa a ser fragmentada, os cálculos das equações da seção anterior representam duas fases de um processamento em *pipeline*.

É importante ressaltar que o cálculo do θ_1 depende do cálculo do θ_2 , caracterizando uma dependência de dados que é comum entre as fases do pipeline.

Figura 6: Distribuição de dados no *pipeline*



2.3 Cpar: uma linguagem de programação para sistemas com processamento paralelo

A linguagem Cpar foi projetada visando oferecer construções simples para exploração do paralelismo em múltiplos níveis, e permitir a otimização do uso da localidade de memória. Em computadores com arquiteturas que apresentam urna hierarquia de memória, tais como sistemas com multiprocessadores com memória compartilhada e memória local, ou sistemas com aglomerados de multiprocessadores com memória compartilhada local em cada aglomerado e uma memória compartilhada global, a exploração da localidade de memória é um aspecto importante na obtenção do alto desempenho.

A linguagem Cpar é urna extensão da linguagem C, na qual foram acrescentadas construções para expressar o paralelismo. Em seu projeto algumas características tiveram a sua origem baseada na linguagem Concurrent C e na linguagem ADA, que oferecem um modelo de programação multitarefas com passagem de mensagem. Paralelizar um programa é distribuir o seu trabalho entre os processadores disponíveis, através da sua partição em múltiplas tarefas que podem ser

executadas simultaneamente. O modelo de programação suportado pela linguagem Cpar permite o uso de múltiplos níveis de paralelismo (ver a figura 7).

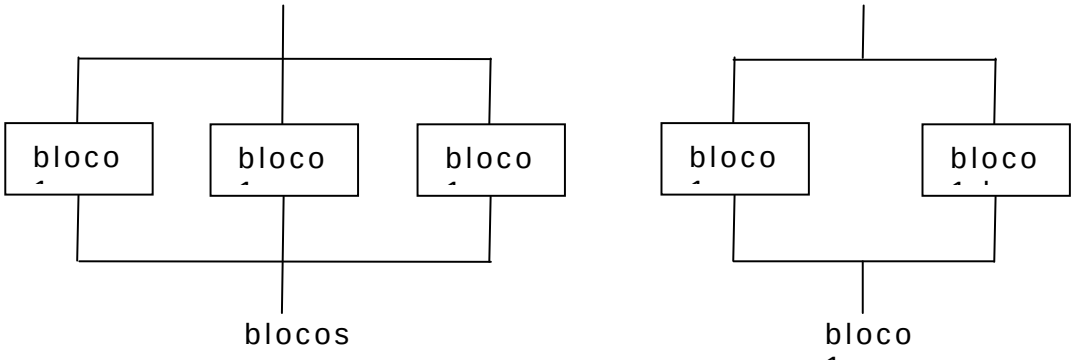


Figura 7: Modelo de programação na linguagem CPar

Blocos contendo elementos, que devem ser executados sequencialmente, podem ser executados simultaneamente. Cada elemento de um bloco pode ser um bloco, promovendo assim, múltiplos níveis de paralelismo. Esta é uma paralelização de granularidade grossa.

O modelo de programação adotado na linguagem Cpar oferece a paralelização da função principal ("main") em múltiplos níveis através dos blocos paralelos. A figura 7 mostra esta característica do modelo.

2.4 Implementação em Linguagem Cpar

O algoritmo descrito anteriormente foi implementado através da linguagem Cpar. Dentre os recursos utilizados estão a criação de processos e utilização de variáveis compartilhadas.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sysitime.h>
#include <unistd.h>
#define L1 200
#define L2 200
shared double x[500000];
shared double y[500000],
shared double ql[500000];
shared double q2[500000];
shared int numElem;

/* Calcular o angulo q1 */
task spec calcularQ1 (numPontos);
task body calcularQ1 (numPontos);
int numPontos;
{int i =0; for (i = 0; i < numPontos; i++){q2[i] =
acos ((x[x[i] + y[i]*y[i] - l1*l1 - l2*l2) / (2*l1*l2));
numElem++;}}
/* Calcular o angulo q2 */
task spec calcularQ2 (numPontos);
task body calcularQ2 (numPontos);
int numPontos,
{int i =0; for (i = 0; i < numPontos; i++)
{ /* Aguardar caso não exista dado da fase
while (numElem < 1); numElem - -;ql[i] =
atan(y[i]/x[i]) – atan(l2*sin(q2[i])) / (l1 + l2 *
cos(q2[i])));}}void main (argc, argv) int argc;
char *argv[ ],
{struct _IO_FILE *arquivo;
   struct timeval tempoInicial;
   struct timeval tempoFinal;
   struct timezone infozone;
   int continuar, i = 0, numPontos = 0;
   char ponto[4];
   /* Verificar se o nome do arquivo de pontos
foi digitado */
   if (argc != 2) {printf ("Digitar: %s <nome do
arquivo> \n", argv[0]); exit (1);}
```

```
 /* Abrir o arquivo */
   arquivo = fopen (argv[1], "rb");
   if (NULL == arquivo) {printf ("Arquivo
inexistente: %s \n",argv[1]);
exit (1);}
   /* Ler os pontos do arquivo */
   for (i = 0; 0 == feof (arquivo); i++)
   {continuar = fread (ponto, sizeof(char), 3,
arquivo);
if (EOF != continuar) {
ponto[3] = O;
x[i] = (double) atoi (ponto);
continuar = fread (ponto, sizeof(char), 3,
arquivo);
if (EOF != continuar)
{ponto[3] = O;
y[i] = (double) atoi (ponto);
numPontos++;} else
perror ("File Format");}}
fclose (arquivo);
numElem = 0; /* nenhum elemento da fase 1
foi calculado */
gettimeofday (ktempoInicial, &infoZona); /*
inicia o timer */
 /* Criar os processos de calculos (fases do
pipeline) */
 alloc_proc (2); /* alocar 2 processadores */
 create 1, calcularQ1 (numPontos);
 create 1, calcularQ2 (numPontos);
 /* Esperar o processo de soma */
 wait_proc (calcularQ1);
wait_proc (calcularQ2);
gettimeofday (&tempoFinal, &infoZona); /*
finaliza o timer */
printf (".\ umero de pontos: %d \n",
numPontos);
printf ("Tempo gasto %d useg. \n",
tempoFinal.tv_sec * 1000000 +
tempoFinal.tv_usec - tempoInicial.tv_sec *
1000000 – tempoInicial.tv_usec);}
```

2.5 Resultado Comparativo

Para obtenção dos pontos de referência da trajetória a ser simulada pelo manipulador, foi elaborado um software (já existe este software) para realizar a especificação dos pontos cartesianos. Para se realizar um estudo comparativo do algoritmo paralelo, foram realizadas a execução sequencial do cálculo de controle e a execução do algoritmo paralelo (algoritmo de pipeline). Os tempos obtidos podem ser observados na tabela e gráfico a seguir.

Numero de Pontos	Tempo Paralelo	Tempo Sequencial
20000	84630	113861
40000	160985	221674
60000	237420	336987
80000	312130	443850
100000	389224	555770
120000	465888	665387
140000	544273	778947
160000	617889	887720
180000	693710	1018953
200000	770606	1107124

Figura 8: Tempo de Execução Sequencial e Paralelo

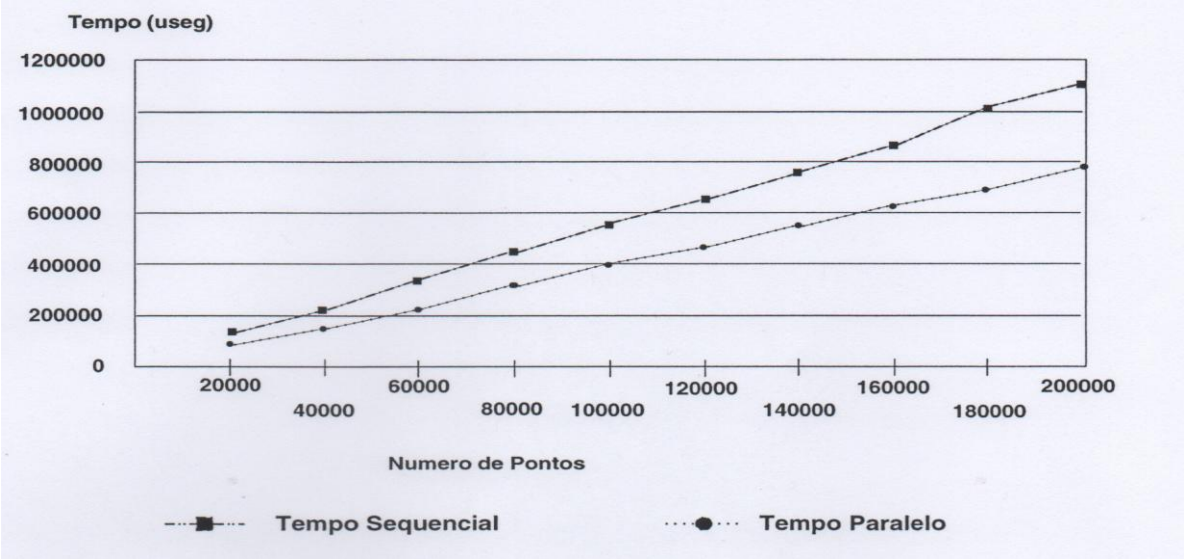


Figura 9: Gráfico de comparação

3 Conclusões

Os recursos de programação paralela presentes na linguagem Cpar possibilitaram a implementação do algoritmo de pipeline no controle de manipuladores. Os tempos obtidos apresentam um desempenho razoável para o algoritmo paralelo. O tempo paralelo obtido não foi exatamente a metade do tempo sequencial devido ao overhead de sincronização entre as fases do pipeline e o gerenciamento dos processos. O *overhead* foi significativo porque a granularidade presente em cada fase é pequena.

O algoritmo descrito pode ser melhorado aplicando simultaneamente outras técnicas de programação paralela, como laços paralelos na fase 1. O pipeline não diminui o tempo de resposta de cada dado que entra no sistema de controle, entretanto o algoritmo possibilita uma maior frequência de entrada dos dados.

Referências

[1] JáJá, J. An Introduction to Parallel Algorithm,s University of Maryland, 1992.
[2] In-Kyu Kim; Teci G. Lewis; Hesham EL-Rewini Introduction to Parallel Computing. Prentice-Hall, 1992
[3] Stadler, W. Analytical Robotics and Mechatronics 1995.
[4]Nomiyama, D.H.; Zorzo, S.D.; Akamatu, D.M. Computação Concorrente: Conceitos e Linguagens de Programação São Carlos. 73p. Relatório de iniciação Cientifica -Departamento de Computação - Universidade Feral de São Carlos.