**Binding of a multiphase schedule to Linear Architecture**

Aryabarta Sahu

Indian Institute of Technology

Guwahati  
Assam, India

asahu@iitg.ernet.in

Sahil Kumar

Indian Institute of Technology

Guwahati  
Assam, India

sahil.kumar@iitg.ernet.in

Nitesh Singhal

Indian Institute of Technology

Guwahati  
Assam, India

s.nitesh@iitg.ernet.in

**ABSTRACT**

Modern processors contain multiple cores. These multicore system enables us to execute multiple applications concurrently. These applications may require more than one phase for their execution. The processor requirement for each phase can vary for an application. These applications are scheduled and binded on this multicore system for execution. So good scheduling and binding strategy can help in achieving high performance and increasing overall throughput of the system.

<previous work>

In this paper, we proposed {5} new approaches for binding of multi-phase applications on multicore linear architecture. Proposed approaches use bipartite matching, biggest block creation etc. to bind applications onto linear multicore architecture. These approaches bind the applications on nearby processors and hence reduces the overall data movement and thus providing speedup. We demonstrate the results using dataset scaling to real life processor requirements.<increase>

**GENERAL TERMS**

Algorithms, Design, Performance, Experimentation

**KEYWORDS**

Multicore processor systems, Linear Architecture, Bipartite matching, Biggest block creation

**INTRODUCTION**

**1. MULTICORE ARCHITECHTURES**

    Latest processors come with large number of cores. Many cores are embedded on a single silicon chip. The cores are interconnected in various network topologies or multicore architectures. Figure a) shows the different multicore architectures. Linear architecture, mesh architecture, hypercube architecture and star architecture are common these days. Multiprocessing, speedup and high throughput can be achieved using multicore systems. These systems run multiple instructions at the same time and executes large number of applications in parallel.

**2. MULTIPHASE APPLICATIONS**

  Most of applications run time characteristics exhibit time varying phase behavior [1, 2]. Applications exhibit different performance metrics during their execution. The various performance metrics can be ILP (Instruction level parallelism), IPC (Instructions per clock period), Hits (number of L1 cache hits), etc. The "phase" of an application is the time interval during which value of the considered performance metrics remains almost constant. So the whole program can be expressed in terms of any performance metric to recognize the various program execution phases. As an example, in [3], Banerjee et. al. detected different phases of applications as shown in Table I

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **SPEC INT** | **Phases** | **SPEC FP** | **Phases** | **MEDIA** | **Phases** |
| **bzip2** | 6 | **ammp** | 4 | **mpeg2enc** | 3 |
| **Crafty** | 3 | **applu** | 3 | **mpeg2dec** | 3 |
| **Gcc** | 6 | **facerec** | 5 | **Cjpeg** | 4 |
| **Gzip** | 6 | **equake** | 6 | **swim** | 3 |
| **Mcf** | 6 | **mgrid** | 4 | **Lame** | 3 |
| **Parser** | 7 | **mesa** | 3 | **Caudio** | 2 |

NUMBER OF PHASES OF BENCHMARKS

**3. APPLICATION SCHEDULING AND BINDING**

A multiphase application may have variable core requirement at various phases of its execution. If the resources are allocated without considering this variable core requirement at different phases of the application then the system will remain underutilized. Thus, scheduling and binding strategy considering multiphase behavior plays an important role in achieving high performance and throughput. Scheduler decides the subset of tasks, the multiphase schedule of applications which are going to be executed on the cores. After the multiphase multiprocess schedule is generated the processor binding algorithm comes into picture. Processor binding algorithm maps or binds the scheduled processes on the multicore system. This algorithm specifies the binding location of the processor on which the task to be binded. This binding is done so that the same application’s tasks are binded nearby. If the cores assigned to the same application are spread far away from each other then there will be more communication delay. So the algorithm should try to reduce the communication delay.

Other consideration while binding processed is to reduce the internal and external fragmentation. Internal fragmentation occurs if more number of the cores are allocated to a process than the number of cores required. External fragmentation occurs if there are enough number of free cores to bind the task but the free cores are not continuous thus resulting in far away binding of the application increasing communication delay.

The main contributions of this paper are (a) a new era of scheduling and binding multi-phase applications on to mesh multicore architecture, (b) surveyed of scheduling multi-phase application (parallel chain) on to multicore architecture and their complexity, (c) Modeling task binding problem in information theoretic model, geometric model and sequence alignment model to understand the problem from different prospective and (d) also we have proposed mixed of critical path and dynamic programming methodology for scheduling multi-phase application on to multicore, and to bind scheduled application to multicore mesh architecture a viable and easy mixed of sequence alignment model and geometric model is proposed and used.

<**Contribution**>

<**Organization**>

<**Motivation**>

**ASSUMED SYSTEM AND APPLICATION MODEL**

**A. System model**

In this paper, we are working on linear architecture of processors. There are N processing elements (PEs or cores or processors)(c0 c1,c2 ...cn-1) interconnected with each other linearly. The assumed communication infrastructure consists of a data network and a control network, each containing routers and channels connected to the cores via standard network interfaces (NIs) as described in [4].<think>

**B. Application model**

All applications are composed of multiple phases. Each phase of an application may require more than one processor. Each phase has two parameters associated with it. One is number of processors that application requires in the phase and other is processing time to complete the phase. As shown in Figure 2 application A0 requires 7 processors for 4 time units in phase one, requires 4 processor for 3 time units in phase 2. Application A0 has 3 phases, A1 has 4, A2 has 2 and A 3 has 5 phases. The arrow represents the dependency among phases. Next phase cannot start before the previous phases have finished execution.

  FIG: Multi-phase running behavior of applications

Let us assume that the schedule consists of set of M applications {a0, a1, a2,.., aM−1 } and K phases {t0, t1 .. tK-1 }. The core requirement for application ai at tj phase is pij. The time to complete for each phase of all applications is same. The total number of cores requirement of any phase for all applications is less than or equal to the total number of available cores.

**DATA MOVEMENT FACTOR**

<If the binding is given then we want to assign the processors where the applications in the current phase will write to and the next phase they will read from and what will be the data movement factor for the given binding.>

<decide figure AND EXAMPLE>

At the start of each phase, cores allotted to each application read data from a specific core where the same application has written data in previous phase. At the end of each phase, cores allotted to each application write data to a specific core from where the cores of same application will read data in next phase. So the data movement overhead in each phase is defined as the sum of above both data reading and writing overheads in that phase. So total Data movement factor (or overhead) is defined as sum of all data movement overheads in each phase.

In each phase, the cores of the same application writes

Let’s consider ith phase in which application A is binded at locations {l1,l2,..,lm} and in (i+1)th phase the same application is binded at {l'1,l'2...,l'n}

Now the cores of application in ith phase write at location x (say) from which the cores of same application in (i+1)th phase read from. So we are minimizing the sum of squares of data movement.

p=no of cores

li = [0,p-1]

l'i = [0,p-1]

dmf=summision((x-li)^2)+summision((x-l'i)^2;

Differentiating this we will get

x=ceil(summision(li)+summision(l'i))/(m+n);

Substituing x in dmf eq we will get dmf of that phase.

For the last phase there won't be any reading overhead of next phase so dmf in that case will become:

x=ceil(summision(li);

dmf=summision((x-li)^2);

So total dmf=summision(dmfi);

**ENTROPY**

<decide figure AND EXAMPLE>

It is a messure of how closely the cores alloted to the same application are binded in the same phase and neighbouring phases. We define total entopy of the binding by the sum of the core entropies of all the phases.

Entropy of jth core in ith phase:

cell\_entropy[i][j]= 1{bind[i][j], bind[i][j+1]} + 1{bind[i][j], bind[i][j-1]} + 1{bind[i][j], bind[i+1][j]} + 1{bind[i][j], bind[i-1][j]}

Total Entropy of all the phases:

Bind Entropy = sumi(sumj (cell\_entropy[i][j]))

In this paper, we try to minimize the total overall entropy that is bind entropy.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**PREVIOUS WORK**

As stated above, our problem is the binding of applications on the cores at all the phases. While binding the applications on any phase we need to keep in mind the binding of the applications on the neighboring phases i.e on the next and the previous phases for the minimization of dmf. So for minimization of overall dmf while binding the applications on the current phase we eventually need to consider the binding of applications on all the phases. While doing this for a particular application for a particular phase we may end up leaving some cores which may be filled by some other application so that identical applications are aligned in successive columns. This problem is exactly similar to multiple sequence alignment (MSA) problem. Finding optimal solution of MSA problem is computationally difficult. This problem is already proved to be NP-complete.[7][8]. So our problem i.e. finding an optimal binding of applications on the linear architecture is NP complete.

general schedule+binding concept ...DAG graph...reference

**SCHEDULING AND MAPPING OF MULTI PHASE APPLICATION TO MULTIPROCESSOR**

**1. SCHEDULING**<detail> <formal definition><formal image>:

Scheduling is an important aspect of operating system. By this threads and processes are given access to system resources. These system resources can be processor time, communications bandwidth, memory address ranges etc. This is done to balance load to achieve a target quality of service. It helps in achieving multitasking and multiplexing.

In general, there are multiple processes that needs execution. The processes can communicate with each other. These can share data with each other. Within one process there can be multiple inherent dependencies. A process consist of tasks with precedence constraint is modeled as a DAG (directed acyclic graph). DAG’s nodes represent tasks of the application and directed edge represent communication or dependency between tasks. The execution of these tasks can be divided into various phases and for these phases these tasks can have multi-processor requirement. So for a given set of N processes, these processes are divided into various sub-tasks which form the DAG and this dag is scheduled to run on a multicore system. The schedule fulfils the dependency criteria among the various processes. The general schedule of processes can be seen in figure.

  Figure: Schedule of processes.

**2. MAPPING**<detail><formal definition> <mapping on general architectures> <communication delay><data sharing>

Mapping is another very crucial step in parallel computing paradigm. It directly affects the performance of the parallel computing system.

The objective is to find a mapping that maximizes the throughput of the system. It is a method of arranging applications on cores for their independent execution on multicore platforms.

So given a schedule i.e. the applications with their processor requirements in each phase, mapping of applications means arranging applications on the cores by using any mapping algorithm.

Mapping of applications is largely dependent on the architecture in consideration.

**COMMUNICATION DELAY**

The architecture which has more connected cores will have less communication delay that the one having less connected cores i.e. if the communication distance between cores in one architecture is less than the communication distance between cores in the other architecture than the communication will be fast in first architecture as compared to the other architecture.

For example, Hypercube architecture will have less communication delay as compared to linear architecture.

**MAPPING APPROACHES AND OPTIMIZATIONS**

Let the schedule consists of set of m applications {a0, a1, a2,.., am−1 } and k phases {t0, t1 .. tk-1 }. The core requirement for application ai at tj phase is pij.There are n processing elements (PEs or cores or processors) (c0 c1,c2 ...cn-1) interconnected with each other linearly.

**1. RANDOM ALGORITHM**

|  |  |
| --- | --- |
|  | **Procedure: Random Algorithm** |
| **1** | Input: Schedule (sch) of the processes. |
| **2** | Output: Binding(b) and total entropy(e). |
| **3** |  |
| **4** | RANDOM\_ALGO( sch ): |
| **5** | generate a random permutation of applications. |
| **6** | b = binding of the applications serially onto the cores in the generated order in each phase. |
| **7** | RANDOM\_ALGO\_HELPER(b) |
| **8** |  |
| **9** | RANDOM\_ALGO\_HELPER(b): |
| **10** | e = intial entropy of the binding b. |
| **11** | l = large number |
| **12** | for i in [1 to l]: |
| **13** | select randomly a row(r) and two columns(c1,c2). |
| **14** | swap the elements b[r][c1] and b[r][c2]. |
| **15** | e' = new entropy of the binding. |
| **16** | if e' < e |
| **17** | e' = e |
| **18** | else |
| **19** | swap the elements b[r][c1] and b[r][c2]. |
| **20** | output the total entropy e and binding b. |

In this algorithm we minimize the overall entropy of the binding. We start with a random generated binding of processes to the cores. Then we calculate the intial total entropy of the binding. We randomly select a row (i.e phase) and two columns(i.e cores on which applications are bind).We try to swap the application core binding for these chosen applications for that phase and again caclulate the new total entropy(e') of the binding and if this e' is less than e then we accept the change else we return to the previous configuration. This we do for large number of times till the value converges.This way by random swapping we are bound to get caught in some local minima or best global minima for some starting configuration. Time complexity of the algorithm is O(m + k\*n + l). Since m,k,n,l are constants so the overall time complexity is O(1). The new entropy calculation can be done in O(1) time by exploiting the relation between the new entropy and the previous entropy.

**2. LEFT RIGHT ALGORITHM**

|  |  |
| --- | --- |
|  | **Procedure: Left-Right Algorithm** |
| **1** | Input: Schedule (sch) of the processes. |
| **2** | Parameter(param) for the cost analysis |
| **3** | Output: Binding(b) and total entropy(e). |
| **4** | LEFT\_RIGHT\_ALGO(sch,param): |
| **5** | list bb = BIGGEST\_BLOCK(sch); |
| **6** | list sp = GENERATE\_SPAN(bb); |
| **7** | bind b; |
| **8** | [rem\_sch,free\_sppace,b ]= phase\_one(sch,sp,b); |
| **9** | b = phase\_two(rem\_sch,free\_space,b,param) |
| **10** | e = calculate total entropy of b |
| **11** | return b, e |
| **12** |  |
| **13** | Input: sch schedule of applications |
| **14** | sp span list of the applications |
| **15** | b binding of the elements |
| **16** | Output: rem\_sch remaining schedule left yet to be binded |
| **17** | free\_space it maps the coninuous free space between the two binded applications |
| **18** | b binding |
| **19** | phase\_one(sch,sp,b): |
| **20** | prev = 0; |
| **21** | for each elem in the sp: |
| **22** | elem.start = prev+1 |
| **23** | elem.end = elem.start + elem.height; |
| **24** | prev = elem.end |
| **25** |  |
| **26** | for i in sp.size(): |
| **27** | if i%2==0: |
| **28** | bind the application on the cores from sp[i].start i.e. from left to right |
| **29** | in each phase according to schedule not overshooting the sp[i]end. |
| **30** | else |
| **31** | bind the application on the cores from sp[i].end i.e. from right to left |
| **32** | in each phase according to schedule not overshooting the sp[i].start. |
| **33** | free\_space = free\_cores in the bind in the each phase. |
| **34** | rem\_sch = schedule remaining after binding initially. |
| **35** | return rem\_sch, free\_space, b |
| **36** |  |
| **37** | Input: rem\_sch remaining schedule left yet to be binded |
| **38** | free\_space it maps the coninuous free space between the two binded applications |
| **39** | b binding |
| **40** | Output: b final binding of applications on the cores |
| **41** | phase\_two(rem\_sch,free\_space,b,param): |
| **42** | alpha = param |
| **43** | beta = 1-param |
| **44** | while free\_space[:][:] != 0: |
| **45** |  |
| **46** | for each app in rem\_sch: |
| **47** | for each space in free\_space: |
| **48** | value[app][space] = 0 |
| **49** | for phase in range [1,k]: |
| **50** | if space[phase] - app[phase] >= 0 |
| **51** | value[app][space] += alpha\*(space[phase] - app[phase]) |
| **52** | else |
| **53** | value[app][space] += beta\*(app[phase]-space[phase] ) |
| **54** |  |
| **55** | assign the not alloted application with minimum value to the free\_space |
| **56** | bind the applications alloted to the the free\_space on the cores according to the schedule for each phase |
| **57** | update free\_space, bind, rem\_sch |
| **58** | return b |
| **59** |  |
| **60** | Input: Schedule (sch) of the processes. |
| **61** | Output: list of the rectangles |
| **62** | BIGGEST\_BLOCK(sch): |
| **63** | list l = {} |
| **64** | for i in [ 1 to m]: |
| **65** | while sch[1:k][i]!=0 : |
| **66** | rect = LARGET\_AREA\_RECT(sch[1:k][i]) |
| **67** | l.insert(rect) |
| **68** | for j in [1:k] |
| **69** | sch[j][i] = (sch[j][i] - rect.height) >=0 ? (sch[j][i] - rect.height) : 0 |
| **70** | sort list using rect area |
| **71** | return list |
| **72** |  |
| **73** | Input: hist is the array containing the core requirement for the phase of an application. |
| **74** | Output: rect with largest area |
| **75** | LARGEST\_AREA\_RECT(hist): |
| **76** | rect = rectangle with largest area in hist such that width(phase) > height(core requirement). |
| **77** | return rect |
| **78** |  |
| **79** | Input: bb is list of rectangles sorted wrt area |
| **80** | Output: span list that spans the cores |
| **81** | GENERATE\_SPAN(bb): |
| **82** | list sp = {} |
| **83** | for each rect in bb: |
| **84** | sum = total sum of the height of all elements in sp |
| **85** | remaining = n - sum |
| **86** |  |
| **87** | if rect.id does not exists in sp |
| **88** | if rect.height <= remaining |
| **89** | insert rect in sp with height = rect.height |
| **90** | else |
| **91** | insert rect in sp with height = remaining |
| **92** | break |
| **93** | else |
| **94** | if rect.height <= remaining |
| **95** | increase the height of the elem with rect.height |
| **96** | else |
| **97** | increase the height of the elem with remaining |
| **98** | break |
| **99** | return sp |

**3.** **CENTER CENTER ALGORITHM**

   Input: Schedule (sch) of the processes.

         Parameter(param) for the cost analysis

  Output: Binding(b) and total entropy(e).

|  |  |
| --- | --- |
|  | **Procedure: Center-Center Algorithm** |
| **1** | CENTER\_CENTER\_ALGO(sch,param): |
| **2** | list bb = BIGGEST\_BLOCK(sch); |
| **3** | list sp = GENERATE\_SPAN(bb); |
| **4** | bind b; |
| **5** | [rem\_sch,free\_sppace,b ]= phase\_one\_center(sch,sp,b); |
| **6** | b = phase\_two(rem\_sch,free\_space,b,param) |
| **7** | e = calculate total entropy of b |
| **8** | return b, e |
| **9** |  |
| **10** | phase\_one\_center(sch,sp,b): |
| **11** | prev = 0; |
| **12** | for each elem in the sp: |
| **13** | elem.start = prev+1 |
| **14** | elem.end = elem.start + elem.height; |
| **15** | prev = elem.end |
| **16** |  |
| **17** | for i in sp.size(): |
| **18** | if i%2==0: |
| **19** | bind the application on the cores from (sp[i].start + sp[i].end)/2 i.e. from middle |
| **20** | and extending both sides i.e. to start and end and not overshooting them in each phase. |
| **21** | else |
| **22** | bind the application on the cores from sp[i].end i.e. from right to left |
| **23** | and extending both sides i.e. to start and end and not overshooting them in each phase. |
| **24** |  |
| **25** | free\_space = free\_cores in the bind in the each phase. |
| **26** | rem\_sch = schedule remaining after binding initially. |
| **27** | return rem\_sch, free\_space, b |

**4. BIPARTITE ALGORITHM**

|  |  |
| --- | --- |
|  | **Procedure: Bipartite Algorithm** |
| **1** | def calculateWeight(i,j): |
| **2** | avgI=Average cores requirement of application i |
| **3** | avgJ=Average cores requirement of application j |
| **4** | weight=0 |
| **5** | S[k][i] = cores requirement of application i in kth phase |
| **6** | for k in range(0,phaseCount): |
| **7** | weight+=(S[k][i]-avgI)\*(S[k][j]-avgJ) |
| **8** | return weight |
| **9** |  |
| **10** | def Bipartite\_Algo(Schedule): |
| **11** | while(No\_Of\_Applications\_in\_Schedule!=1): |
| **12** | for all (i,j) pair: |
| **13** | Edge\_Weight\_Between\_i\_and\_j=calculateWeight(i,j) |
| **14** | Construct graph G={applications,edges} |
| **15** |  |
| **16** | perfect\_matched\_edges=Blossom\_Algorithm(G) |
| **17** | new\_schedule=merge schedule corresponding to the applications i & j if there is edge between i & j in perfect\_matched\_edges |
| **18** | Store the perfect\_matched\_edges to be used later |
| **19** |  |
| **20** | binding\_pattern=Finally the stored information will give the sequence in which the of applications should be binded.={ABCDEFGH} |
| **21** | pairs={AB,CD,EF,GH} |
| **22** |  |
| **23** | for each (i,j) in pairs: |
| **24** | maxpairReq[ij]=Maximum\_processors\_needed to bind p.i application as well as p.j application |
| **25** |  |
| **26** | for (i,j) in pairs: |
| **27** | initialBinding=initialBinding + (bind ith application from left and jth application from right in maxpairReq[ij] processors in each phase) |
| **28** |  |
| **29** | extraColumns=Number\_Of\_Columns\_Used-Number\_Of\_Processors |
| **30** |  |
| **31** | finalBinding=Remove extraColumns which are least binded and bind those applications the required cores linearly |
| **32** |  |
| **33** | return finalBinding |

**EXPERIMENT AND RESULT ANALYSIS**<2 PAGE><GRAPH><AND IT'S EXPLATION><WHY ONE PERFORMS BETTER AND OTHER DOESN'T ><IMPLEMENTATION DIFFERENCE AND WHY ONE ALGO IS BETTER><INERPRETATION OF CONCLUSION>:

**CONCLUSION AND FUTURE WORK**

**REFERENCES**

1. Wang L, Jiang T. (1994). "On the complexity of multiple sequence alignment". J Comput Biol 1 (4): 337–48. doi:10.1089/cmb.1994.1.337. PMID 8790475.

2. Jump up ^ Elias, Isaac (2006). "Settling the intractability of multiple alignment". J Comput Biol 13 (7): 1323–1339. doi:10.1089/cmb.2006.13.1323. PMID 17037961.

http://complexhpc.org/events/lisbon/PDF/Runger.pdf for motivation