Comparision of Binding Approaches of Scheduled Multiphase Application to Multicore Architecture

Abstract—***Modern processors contain multiple cores. These multicore systems enable 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 in increasing overall throughput of the system. In this paper, we proposed four new approaches for binding the multi-phase applications on multicore linear architecture. Proposed approaches use bipartite matching, biggest block creation etc. to bind the applications on linear multicore system. These approaches bind the applications on nearby processors and hence reduces the overall data movement and thus providing speedup. We demonstrate the results using data set scaling to real life processor requirements.***

I. Introduction

A. Multicore Architecture

Chip multiprocessor (CMP) exploits the increasing device density in a single chip. Modern processors contain multiple cores which enables them to concurrently execute multiple applications (or tasks) on a single chip. More than thousand cores on a single chip are common now. In our context **large chip multiprocessors (LCMP)** means a single chip multiprocessor contains more than 100's or 1000's of processors [1].

Modern processors come with large number of cores. Many cores are embedded in a single silicon chip. These cores are interconnected in various network topologies or multicore architectures. Fig. 1 shows the different multicore architectures. Linear architecture 1(a), Mesh architecture 1(b), Hyper cube architecture 1(c) and Star architecture 1(d) 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 execute large number of applications in parallel.

B. Multiphase Application

Most of applications run time characteristics exhibit time varying phase behavior [2, 3]. Applications impose different performance metric values in different time intervals. During each phase an application has same value for performance metrics. These applications running time can be partitioned in different number of phases according to performance metrics like **Instruction level parallelism (ILP)**, **Instructions per clock period (IPC)**, **Number of L1 cache hits (Hits)** etc. As an example, in paper [4], Banerjee et. al. used IPC profile to detect different phases of applications running time. They observed different benchmarks program and detected phases as shown in Table I.

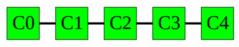
SPEC INT	Phases	SPEC FP	Phases	MEDIA	Phases
bzip2	6	ammp	4	mpeg2enc	3
crafty	3	applu	3	mpeg2dec	3
gcc	6	facerec	5	cjpeg	4
gzip	6	equake	6	swim	3
mcf	6	mgrid	4	lame	3
parser	7	mesa	3	caudio	2

TABLE I
Number of phases of benchmarks [4]

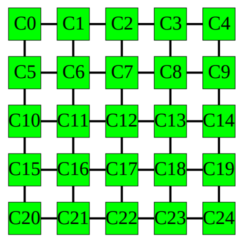
C. Application scheduling and binding

A multiphase application may have variable core requirement at various phases of its execution. If the resources are allocated without considering the 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 binding algorithm comes into picture. The processor binding algorithm maps or binds the scheduled applications on the multicore system. This algorithm specifies the binding location of the core 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.

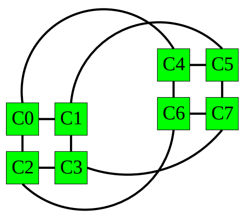
Other consideration while binding applications is to reduce the internal and external fragmentation on cores. Internal fragmentation occurs if more number of the cores are allocated to an application than the number of cores required. External fragmentation occurs if there are enough number of free cores to bind the task but the free cores are not continuous, thus



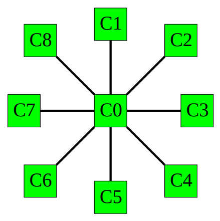
(a) **Linear Array**



(b) **Mesh Architecture**



(c) **Hyper cube Architecture**



(d) **Star Architecture**

Fig. 1. **Multicore interconnection network**

resulting in distant binding of applications and increasing communication delay.

D. *Contribution and Organization*

TODO

II. Assumed Model and Architecture

A. *Architecture model*

In this paper, we are working on cores arranged in linear architecture 1(a). There are n interconnected **processing elements(PEs or cores)** c_1, c_2, ..., c_n. 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 (NI)** as described in [5].

B. *Assumed application model*

The applications span over multiple phases. Each phase of an application can be divided into one or more tasks. Each phase of an application may require more than one core. Each phase has two parameters associated with it. The parameters associated are number of cores that application requires in the phase and processing time to complete the phase. As shown in Fig. 2 application a_1 requires 7 cores for 4 time units in phase t_1, a_2 requires 4 cores for 3 time units in phase t_2. Applications a_1, a_2, a_3, a_4 has 3, 4, 2 and 5 phases respectively. The arrow represents the dependency among phases. Next phase cannot start before the previous phase has finished execution.

Let us assume that the schedule consists of set of m applications $\{a_1, a_2 \ldots, a_m\}$ and k phases $\{t_1, t_2 \ldots, t_k\}$. We are given the schedule sch of the applications. The core requirement in t_i phase for application a_j is sch_{ij}. The time to complete for each phase of all applications is assumed same. The total number of core requirement for any phase for all applications is less than or equal to the total number of

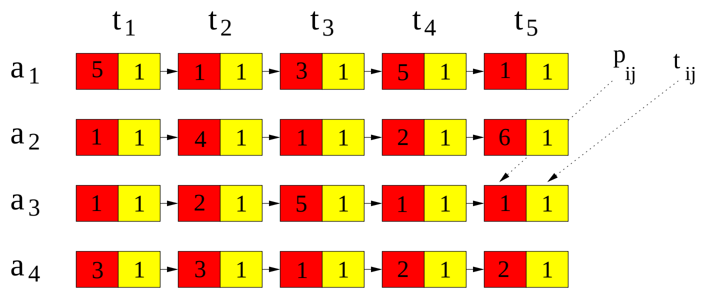


Fig. 2. Multi-phase running behavior of applications

available cores. The schedule sch is mapped on the cores to get the binding b. The application binded in phase t_i on core c_j is represented by b_{ij}.

C. *Allocation model*

The main goal of mapping is to allocate the appropriate resources (number and relative position of cores to be allocated) to the incoming application tasks such that the communication cost and memory access cost is minimized. At the same time, all the pre-existing applications still continue to run on the initial set of resources they have been allocated earlier.

1) Data movement factor (dmf): This is a measure of overall movement of data during execution of the scheduled applications on multicore system.
The applications are binded on the cores in each phase. At the start of phase, cores allotted to an application read data from a specific core where the same application has written data in the previous phase. At the end of phase, cores allotted to each application chooses a core and write data to it from where the cores of the same application will read data in next phase.
The data movement overhead for a phase is defined as the sum of both reading overhead and writing overhead for that phase. So total dmf is defined as sum of all data movement overheads in each phase for all the applications.

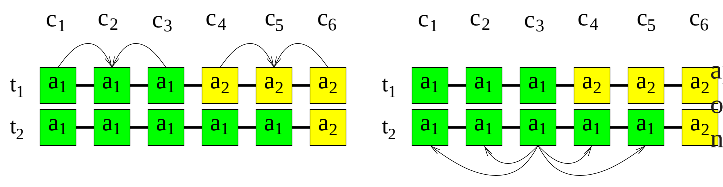


Fig. 3. Data Movement Factor

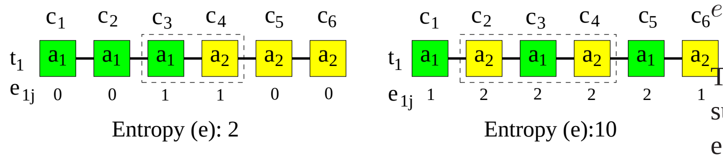


Fig. 4. Entropy

Let us consider ${t_i}^{th}$ phase in which application a_j is binded at locations $\{l_1, l_2 \ldots, l_u\}$ and in ${t_{i+1}}^{th}$ phase the same application is binded at $\{l'_1, l'_2 \ldots, l'_v\}$. Now the cores of application in ${t_i}^{th}$ phase writes at location x_i from which the cores of same application in ${t_{i+1}}^{th}$ phase read from. So we are minimizing the sum of squares of data movement.

$$dmf_i = \sum_{j=1}^{u}(x_i - l_j)^2 + \sum_{j=1}^{v}(x_i - l'_j)^2 \quad (1)$$

$l_i \in 1, 2 \ldots, n$
$l'_i \in 1, 2 \ldots, n$
and differentiating this we will get,

$$x_i = \lceil (\sum_{j=1}^{u} l_i + \sum_{j=1}^{v} l'_j)/(u+v) \rceil \quad (2)$$

Substituting x_i from 2 in equation 1 we will get dmf_i of phase t_i. For phase t_k there won't be any writing overhead of next phase so dmf_k and x_k in that case will become

$$dmf_k = \sum_{j=1}^{u}(x_k - l_j)^2 \quad (3)$$

$$x_k = \lceil (\sum_{j=1}^{u} l_i)/u \rceil \quad (4)$$

Similarly there will be no reading overhead for t_1 phase and dmf_1 can be similarly calculated. Therefore, the total data movement factor dmf can be calculated as

$$dmf = \sum_{i=1}^{k}(dmf_i) \quad (5)$$

2) Entropy: It is a measure of closeness of tasks of an application binded on multicore system. This measure depends on the binding of tasks of an application in both same and neighboring phases.

Core Entropy e_{ij} in phase t_i of core c_j is defined as the number of tasks assigned on neighboring cores which differs from the task assigned on the current core. Thus,

$$e_{ij} = 1\{b_{i,j}, b_{i,j+1}\} + 1\{b_{i,j}, b_{i,j-1}\} + 1\{b_{i,j}, b_{i-1,j}\} + 1\{b_{i,j}, b_{i+1,j}\} \quad (6)$$

Therfore the total entopy e of the binded applications is the sum of core entropies of all the cores in all phases. The equation becomes

$$e = \sum_{i=1}^{k}\sum_{j=1}^{n} ce_{ij} \quad (7)$$

III. Previous Work

As stated above, our problem is binding of applications on the cores in each phase. While binding the applications on phase t_i we need to keep in mind the binding of the applications on the previous and the next phases for the minimization of data movement. So for minimization of overall data movement 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 [6][7][8]. So our problem i.e. finding the optimal binding of applications on the linear architecture in multi-phase system is NP complete.

IV. Scheduling of multiphase application on to multiprocessor

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 applications that needs execution. These applications can communicate with each other. These can share data with each other. Within one application there can be multiple inherent dependencies. An application consist of tasks with precedence constraint is modeled as a **directed acyclic graph (DAG)**. 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 each phase these tasks can have multicore requirement. So for a given set of m applications, these applications are divided into various subtasks which form the DAG which is scheduled to run on a multicore system. The schedule fulfill the dependency criteria among the various processes. The schedule of processes can be seen in Fig. 2.

V. Mapping of scheduled multiphase application on to multiprocessor

A. Mapping

Mapping is another very crucial step in parallel computing paradigm. It directly effects 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 core requirements in each phase, mapping of applications means arranging applications on the cores by using a mapping algorithm. Mapping of applications is largely dependent on the architecture in consideration.

1) Communication Delay: The architecture which has more connected cores will have less communication delay than 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 then the communication will be fast in first architecture as compared to the other architecture. For example, Hyper cube architecture will have less communication delay as compared to linear architecture.

2) Mapping Approaches and Optimization:

VI. Mapping Algorithms

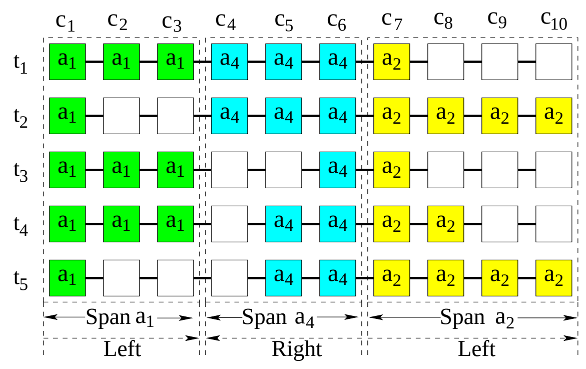
A. Random Algorithm

As shown in algorithm 1, we minimize the overall entropy of the binding. We start with a random generated binding of processes to the cores. Then we calculate the initial 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 calculate 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 and 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.

Procedure: 1 Random Algorithm for Mapping

Input: Schedule (sch) of the processes
Output: Binding(b) and total entropy(e).

```
1: Generate a random permutation of applications.
2: $b$ = Binding of the applications serially onto the cores in the generated
   order in each phase.
3: $e$= Initial entropy of the binding $b$.
4: $L$ = Large number
5: for $i = 1$ to $L$ do
6:    Select randomly a row(r) and two columns(c1,c2).
7:    Swap the elements b[r][c1] and b[r][c2].
8:    $e'$ = new entropy of the binding.
9:    if $e' \leq e$ then
10:       $e' = e$
11:   else
12:       Swap the elements $b[r][c1]$ and $b[r][c2]$.
13: Output the total entropy $e$ and binding $b$.
```

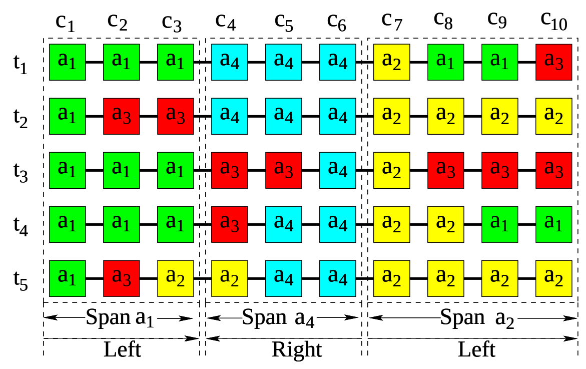


c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10}
t_1 t_2 t_3 t_4 t_5
a_1 a_4 a_2
Span a_1
Span a_4
Span a_2
Left
Right
Left

Fig. 5. After Left-Right Coarse Pass Algorithm 5

B. Left-Right Algorithm

This algorithm 2, uses various procedures to bind the applications on the cores. It first recognizes the applications with the highest core demand in the continous phases by calling the Biggest Block algoritm 3. After recognizing the biggest blocks of applications, it calculates the span of the applications (i.e. allocates some number of cores for specific applications for all the phases) by calling the Generate Span algorithm 4. Then we bind the applications in the span list on the cores by calling the Coarse Pass algorithm 5. The binding of the applications is done alternatively from left to right and right to left on the



c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10}
t_1 t_2 t_3 t_4 t_5
a_1 a_2 a_3 a_4
Span a_1
Span a_4
Span a_2
Left
Right
Left

Fig. 6. Fine Pass 6

Procedure: 2 Left-Right Algorithm for Mapping

Input: Schedule sch, Parameter $param$ for the cost calculation
Output: Binding b, Total entropy e

```
1: list $bb$ = Biggest-Block($sch$)
2: list $sp$ = Generate-Span($bb$)
3: ($rem\_sch$, $free\_space$, $b$) = Coarse-Pass($sch, sp, b$)
4: b = Fine-Pass($rem\_sch$, $free\_space$, $b$, $param$)
5: $e$ = calculate total entropy of $b$
6: Output the total entropy $e$ and binding $b$.
```

Procedure: 3 Biggest-Block

Input: Schedule sch
Output: Biggest-Block List bb

```
1: list $bb$
2: for $i$ = 1 to $m$ do
3:     while $sch_{(1:k)i}$ !=0 do
4:         $rect$ = Calculate largest rectangle in array $sch_{(1:k)i}$
5:         Insert $rect$ in $bb$
6:         for each phase $k$ do
7:             Subtract the width of the $rect$ from the $sch$ of the application $i$
8: Sort the list $bb$ according to area and then according to height $h$ or rectagle
9: Return the list $bb$.
```

cores spanned by them according to the span list generated as shown in Fig. 5. According to the Fig. 5, the application a_1 is binded on the cores from the left to right in the span of this application. The next application a_4 is binded from right to left in the span. And similarly application a_2 is binded from left to right.The binding is done to fulfill the maximum schedule requirement of the applications for a particular phase. Then we update the free cores and the schedule of the applications that still needs to be binded. Now, we call the Fine Pass algorithm 6 by passing the required parameters for binding the remaining schedule on the free cores. The applications are binded on the free cores according to the parameter $param$ passed to the function Fine Pass algorithm 6. Then the total entropy (e) of the binding is calculated and binding b and entropy e are returned.

1) Biggest-Block Algorithm: This algorithm takes the schedule sch as input. It breaks the core requirement of an application in the terms of big rectangular blocks whose $height$ is the phases and the $width$ is the core requirement. This way rectangular blocks are formed for each of the application and inserted in the biggest-block list bb. This list bb is sorted according to $area$ and according to $height$ in case of tie and returned by this algorithm.

2) Generate-Span Algorithm: This algorithm 4 takes the list of rectangles bb returned by the Biggest-Block algorithm 3 as input. It takes the elements from the rectangle one by one and inserts into the span list sp. The $rect$ is inserted in list sp only if there is no already existing rectangle in the list sp with same application number as of the $rect$. If there do exists a rectangle in the list sp with same application number as of the $rect$,then the $span$ of the $elem$ in the list sp is increased by the $width$ of the $rect$. While inserting the elements in the list sp, we keep in mind that sum of the span of all the elements in

Procedure: 4 Generate-Span

Input: List of rectangles bb
Output: Span list of applictions sp

```
1: list $sp$
2: for each rect in $bb$ do
3:     if $rect.id$ exists in $sp$ then
4:         if $cores$ left unassigned then
5:             Increase the $cores$ alloted ($span$) of the $rect.id$ in list $sp$ by
               minimum of $cores$ remaining or $rect.height$
6:         else
7:             break
8:     else
9:         Insert the $rect$ in the list $sp$ with $span$ equal to minimum of $cores$
           remaining or $rect.height$
10: Return the list $sp$.
```

Procedure: 5 Coarse-Pass

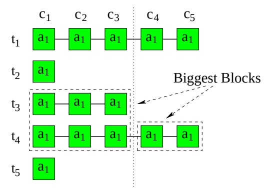
Input: Schedule sch, span list sp
Output: remaining schedule rem_sch, the continuous free space $free\_space$ between the two binded applications, binding b

```
1: $prev$ = 0
2: for each $elem$ in $sp$ do
3:     $elem.start$ = $prev$+1; $elem.end$ = $elem.start$ + $elem.height$
4:     $prev$ = $elem.end$
5: for $i$ = 1 to $size$ of $sp$ do
6:     if i mod 2==1 then
7:         Bind the application on the cores from $sp[i].start$
8:     else
9:         Bind the application on the cores from $sp[i].end$
10:    $free\_space$ = free cores in the bind in the each phase.
11:    $rem\_sch$ = schedule remaining after binding initially.
12: Output $rem\_sch$, $free\_space$ and $b$
```

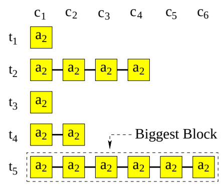
the list sp is less than or equal to the total number of available cores n.

3) Coarse-Pass Algorithm: This algorithm 5 takes schedule of applications sch, span list sp to output the bind the applications b. This algorithm 5 forms the procedure for Left-Right algorithm 2. It binds the applications on the cores, according to their $span$ in span list sp and maximally satisfies the $core$ requirement at each phase. The binding for the applications at odd postion in the span list(sp) is done from left to right starting from the $elem.start$ to $elem.end$ where $elem$ is the element in the span list(sp) as shown in Fig. 5. Similarly for the applications at even position in the span list(sp) the binding is done from right to left starting from the $elem.end$ to $elem.start$ as shown in Fig. 5. The objective of binding this way is to maximize the space in the middle of two consecutive binded applications. This ways the applications can be binded together in the next binding phase by Fine-Pass algorithm 6.After binding the elements to the cores, the schedule sch, binding b and free cores$free_space$ is updated and outputted.

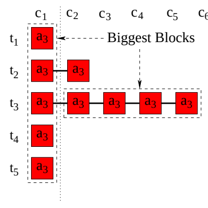
4) Fine-Pass: This algorithm 6 takes remaining schedule rem_sch, continuous free space $free\_space$ between the two binded applications, parameter $param$ and binding b after Coarse-Pass algorithm 5 as input. It calculates the $value_{i,j}$ when the applicationa_i is binded in the $free\_space_j$. This $value_{i,j}$ comprises of the two cost parameters namely undershoot value($u_{i,j}$) and overshoot value($o_{i,j}$).



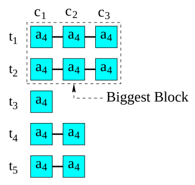
(a) **Biggest Blocks of** a_1



(b) **Biggest Blocks of** a_2



(c) **Biggest Blocks of** a_3



(d) **Biggest Blocks of** a_4

Fig. 7. **Biggest Blocks**

The undershoot value for a phase is the α times the space that remains unallocated while binding the application a_i in the continuous block of $free\_space_j$ at phase t_k. Thus undershoot value $u_{i,j}$ is the summation of the total unbinded space while binding the application(a_i) at the continuus $free\_space_j$.

$$u_{i,j} = \sum_{l=1}^{k} \alpha * (space[k] - app[k]) \tag{8}$$

if ($space[k]$ - $app[k]$ $\geq$ 0 else 0) , here α = $param$ Similarly, overshoot value $o_{i,j}$ for a phase is defined as the β times the schedule($rem\_sch_{i,k}$) that remains unbinded while binding the application a_i in the continuous block of $free\_space_j$ at phase t_k. Thus overshoot value $o_{i,j}$ is the summation of the total unbinded schedule while binding the application a_i at the continuus free space (j).

$$o_{i,j} = \sum_{l=1}^{k} \beta * (app[k] - space[k]) \tag{9}$$

if ($space[k]$ - $app[k]$ $\geq$ 0 else 0) , here $beta = 1 - param$ Therfore, Fitting Cost $fc_{i,j}$ can be calculated as

$$fc_{i,j} = u_{i,j} + o_{i,j} \tag{10}$$

This way we find the cost $fc_{i,j}$ for each application a_i at each $free\_space_j$.For particular $free\_space_j$, the application a_i with minimum value($fc_{i,j}$) is binded on the $free\_space_j$.

Procedure: 6 Fine_Pass

Input: rem_sch remaining schedule left yet to be binded, $free\_space$ it maps the continuous free space between the two binded applications, $param$, and b binding **Output:** b the final binding of applications on cores

```
1:  α = param, β = 1 − param
2:  for each Apps in rem_sch do
3:      for each space in free_space do
4:          value[app][space] = 0
5:          for for phase in range [1,k] do
6:              if space[phase] - app[phase] >= 0 then
7:                  value[app][space] += alpha*(space[phase] - app[phase])
8:              else
9:                  value[app][space] += beta*(app[phase]-space[phase] )
10:             Assign the not alloted application with minimum value to the free_space
11:             Bind the Apps alloted to the the free_space on the cores according to the schedule for each phase
12:             Update free_space, bind, rem_sch
13: Output rem_sch, free_space and b
```

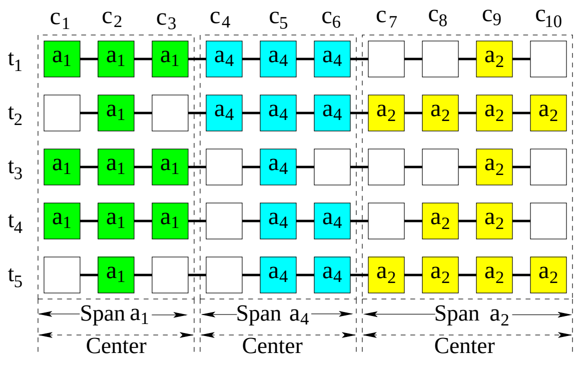


Fig. 8. After Center-Center Coarse Pass

This is done for all the $free\_space$. This whole process is repeated till all the applications are successfully binded on all the $free\_spaces$. After this we output the final binding b.

C. Center-Center Mapping Algortihm

This algorithm 8 is quite similair to the previous algorithm discussed i.e. left-right algorithm 2. The only major difference between the two algorithm comes in the phase one of the binding of the applications on the cores. In left-right algorithm 2, in phase one of binding the consecutive applications were binded in the left to right and right to left fashion. But in this algorithm, the applications are binded within the span it covers on the cores starting from the middle as shown in Fig 8. That is the binding of the application on the cores in the span is middle aligned. This is done for all the applications. After this we update the free cores and calculate the remaining scheule of applications. Then as in the previous algorithm 2, we bind the remaining schedule on the $free\_space$ of cores by the fine-pass algorithm 6. Calculation for the total entropy e is done. Then the entropy value e and binding b are outputted.

1) Coarse Pass Center: This algorithm 8 takes schedule of applications sch, span list sp to bind the applications on the binding b. This algorithm is specific to the Center-Center algorithm 7. This algorithm is similar to the previous

Procedure: 7 Center-Center Algorithm for Mapping

Input: Schedule sch, Parameter $param$ for the cost calculation
Output: Binding b, Total entropy e

```
1: list bb = Biggest-Block(sch)
2: list sp = Generate-Span(bb)
3: (rem_sch, free_space, b) = Coarse-Pass-Center(sch, sp, b)
4: b = Fine-Pass(rem_sch, free_space, b, param)
5: e = calculate total entropy of b
6: Output the total entropy e and binding b.
```

Procedure: 8 Coarse-Pass Center

Input: Schedule sch, span list sp
Output: remaining schedule rem_sch, the continuous free space $free\_space$ between the two binded applications, binding b

```
1: prev = 0
2: for each elem in sp do
3:     elem.start = prev+1; elem.end = elem.start + elem.height
4:     prev = elem.end
5: for i = 1 to size of sp do
6:     Bind the application on the cores in the middle of the span from sp[i].start to sp[i].end fulfilling the scheule sch[i] requirement.
7:     free_space = free cores in the bind in the each phase.
8:     rem_sch = schedule remaining after binding initially.
9: Output rem_sch, free_space and b
```

Coarse Pass 5 used in Left-Right algorithm 2. This algorithm 8 binds the applications on the binding(b). The binding is done according to the information in the span list(sp). The binding of the application is on the cores is center aligned done in the span for that particular application dictated by the span list(sp) as shown in Fig. 8. This is done for all the applications in the span list sp, considering the schedule sch of the application. After the initial phase of binding the new values of remaining schedule rem_sch, free cores $free\_space$ are calulated. This algorithm thus outputs the schedule sch, binding b and free cores $free\_space$.

D. Modified Blossom

The algorithm 10 takes scheduling innformation of applications as input. It first tries to find the application order as well as application pairs such that if the applications are binded using this information will give a better binding. We first find the application pairs. For this we first convert the sheduling information into a graph. In graph, nodes will represent applications and weighted edges represent the compatibility score between two applications. Less the compatibility score will mean more the compatibility. We are using Blossom algorithm for finding minimum cost perfect matching. The

Procedure: 9 Blossom_Weight

Input: Schedule sch of the processes, application a_i, application a_j
Output: Weight w between application a_i and application a_j.

```
1: avgI=Average cores requirement of application i
2: avgJ=Average cores requirement of application j
3: weight=0
4: for p in range(0,k) do
5:     $weight += (sch[p][i] - avgI) * (sch[p][j] - avgJ)$
6: return weight
```

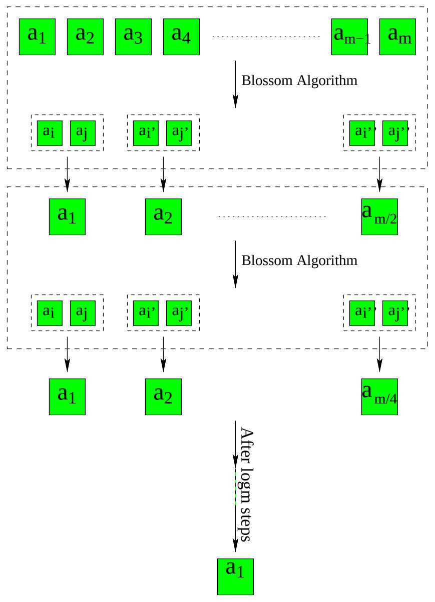


Fig. 9. Modified Blossom

output of blossom algorithm will be the pairs of applications such that total cost is minimized in perfect matching. We store the pair information which we will use for finding the application order. We then modify the scheduling information. For it, we will assume both the applications in pair information as a single application and again find the graph and run Blossom algorithm. We do this until the applications count comes down to 1. Let say the application order we got from above be $\{a_1, a_2 \dots, a_m\}$. And let say pairs P be $\{(a_1, a_2), (a_3, a_4) \dots, (a_{m-1}, a_m)\}$. Then for each pair in P we find the minimum processor requirement $minpairReq$ of them if they are binded toogether and bind that pair in $minpairReq$ cores. Binding is represented as k*n matrix. We will see that we have used more cores i.e. columns in binding matrix than the actual given number. Remove the extra columns in which the binding is least and bind those applications in the phases wherever the cores are not used in linear fashion starting from first column. The new binded matrix after adjusting of columns will be the final binding matrix.

1) Blossom Weight: The algorithm 9 finds the weight between two applications using the scheduling information. We are using this algorithm while making edges between nodes while we are converting the scheduling information into a graph. We find the average cores requirement of a_i^{th} and a_j^{th} application and store it into $avgI$ & $avgJ$ respectively. So the weight between two applications is calculated as product of how much extra both applications require than their average requirement. Weight will be the sum of the above calculated product in each phase.

Procedure: 10 Modified_Blossom

```
Input: Schedule $sch$ of the applications
Output: Binding $b$ and total entropy $e$
1: while $No\_Of\_Applications\_in\_Schedule$!=1 do
2:    for all ($a_i$,$a_j$) pair do
3:       $Edge\_Weight\_Between\_a_i\_and\_a_j$=$Blossom\_Weight(sch, a_i, a_j)$
4:    Construct graph $G$=(applications,edges)
5:    $perfect\_matched\_edges$=$Blossom\_Algorithm(G)$
6:    Store the $perfect\_matched\_edges$ information
7:    new_schedule=after merging schedule corresponding to the applications
      $a_i$ & $a_j$ if ($a_i$,$a_j$) $\in$ $perfect\_matched\_edges$
8: The stored information will give the sequence $\{a_1, a_2 \dots, a_m\}$ in which
   the applications should be binded.
9: $pairs$=$\{(a_1, a_2), (a_3, a_4) \dots, (a_{m-1}, a_m)\}$
10: for ($a_i$,$a_j$) in $pairs$ do
11:    $minpairReq[ij]$=Minimum_cores_needed to bind $a_i^{th}$ & $a_j^{th}$ appli-
       cations together
12:    $initialBinding$=$initialBinding$ + (bind $a_i^{th}$ and $a_j^{th}$ applications
       in minpairReq[ij] alloted cores in each phase)
13: $extraColumns$=$Number\_Of\_Columns\_Used$-$n$
14: $finalBinding$=Remove $extraColumns$ which are least binded and
    bind them linearly
15: return $finalBinding$
```

VII. Experiment and result

(2 PAGE, GRAPH, AND IT'S EXPIATION, WHY ONE PERFORMS BETTER AND OTHER DOESN'T, IMPLEMENTATION DIFFERENCE AND WHY ONE ALGO IS BETTER, INTERPRETATION OF CONCLUSION)
The graphs in the Figure 10, Figure 11 and Figure 12 shows the variation of total entropy with the variation in number of processors, phases and phases by running the various algorithms described in this paper on the multiphase schedule. The red line in the graphs indicate the initial entropy. This value is calulated by binding the multiphase schedule on the cores by generating any random permutation of applications and binding them on the cores in the linear fashion. This is the initial entropy value and is the highest for all the data points. This highest entropy value indicates that the binding of schedule on the cores in not efficient.

The plot of Random Algorithm(1) is shown with the green line. In this algorithm after binding the schedule on the cores in the random manner, we minimzed the entropy value by swapping the applications on the cores for a particular phase. This process of swapping the applications on cores was again done randomly for the phases and for the cores in a particular phase. This random swapping was done for a large number of times accepting the swap if the new entropy value was lower than the previous value. This algorithm leads to the low values of overall total entropy. This algorithm has a drawback i.e. it may not always give the optimal minimum value of total entropy. As the initial starting binding is random and this plays a significant roles in the subsequent random swaps. By this the state may be trapped in the local minima and the configuration of binding never changes. This local minima problem is serious and this is what happens, hence the carefully devised algorithm may give better results than the Random Algorithm 1.

The total entropy value of Left-Right Algorithm(2) is in between the total initial entropy value and the total entropy value after applying Random Algorithm(1). In this algorithm, the binding was done in the two crucial phases. In the phase phase of the binding the biggest continuous rectangular application blocks in the continuous phases were recognized. After recognizing these biggest blocks, span(cores) for the applications with the highest area were allocated till all the cores gets spanned. Then the binding on the spanned cores was done by the applications according to the schedule.This phase may lead to some free cores in some of the phases and schedule of some application unsatisfied. To bind the remaining schedule, we call the Phase-Two Algorithm(6). This algorithm takes the parameter($param$). This parameter is used in calculating the cost values while assigning the free cores to the remaining schedule. By using this $param$ value we calculate the undershoot parameter($alpha$) and overshoot parameter($beta$). The undershoot parameter($aplha$) is responsible for taking in account the penalty for the free cores left(internal fragmentation) while binding the application in the continous free space between the two consecutive applications binded in phase one. The overshoot parameter($beta$) penalizes for the schedule of the application that still is unsatisfied after binding the application in the free space between the two consecutive applications binded in phase one. We want to minimize the overall penalty. The application best-fitted in the free spaces. This results shown in the graphs are when parameters $alpha$ and $beta$ are given equal weightage. This way the remaining schedule is binded to the cores and the overall entropy value is calculated. This algorithm performs slighty better than Center-Center Algorithm(7) but significantly reduces the intial value of the entropy. We can apply the Random Algorithm(1) over the binding we got from this alogrithm as the initial configuratiom. This way we can further minimize the overall total entropy of the binding.

The results of the Center-Center algorithm(7) are comparable to the results of the Left-Right algorithm 2. This algorithm also binds the schedule in the two phases. This algorithm differs from the Left-Right algorithm(2) just in the phase one of the binding. In phase one of the algorithm the binding is done by calling the Phase-One-Center Algorithm(8). It center binds the applications in the cores spanned by them. The binding in second phase is done by calling the Phase-Two algorithm(6). This also considers the

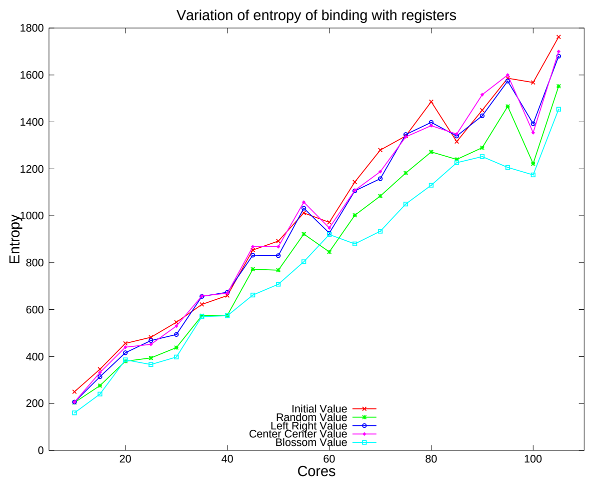


Fig. 10. **Variation of entropy with processors**

undershoot($alpha$) and overshoot parameter($beta$) values while binding. By running Random Algorithm(1) over the binding outputted by this algorithm, the total entropy value can be further improved.

The results of Modified Blossom algorithm(10) is far better than Left-Right algorithm 2 and Center-Center algorithm(7). This algorithm performs better than the random algorithm also. The main crux of the algorithm lies in extracting the application order as well as application pairs such that if the applications are binded using this information will give a good binding. The algorithm uses Blossom Algorithm (provide reference) logm times to find the application order. After finding the application order the binding step is simple. We bind the applications in pair according to the application order that we got from above. Since this algorithm binds considering a more strong form of closeness between applications, this algorithm performs best among the available algorithms.

Figure 10. is shows Variation of entropy with processors. Figure 11. is shows Variation of entropy with number of phases. Figure 12. is shows Variation of entropy with number of application.

VIII. Conclusions and Future Work

The minimum value of total entropy is obtained by running the blossom algorithm to bind the given schedule. Left-Left Algorithm (2) and Center-Center Algorithm(7) gives the average resutls between the initial entropy value and the total entropy value obtained by running the Blossom algorithm.

References

[1] L. Zhao, R. Iyer, S. Makineni, R. Illikkal, J. Moses, and D. Newell, “Constraint-aware large-scale cmp cache design,” in *Proceedings of the 14th International Conference on High Performance Computing*, ser. HiPC’07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 161–171.

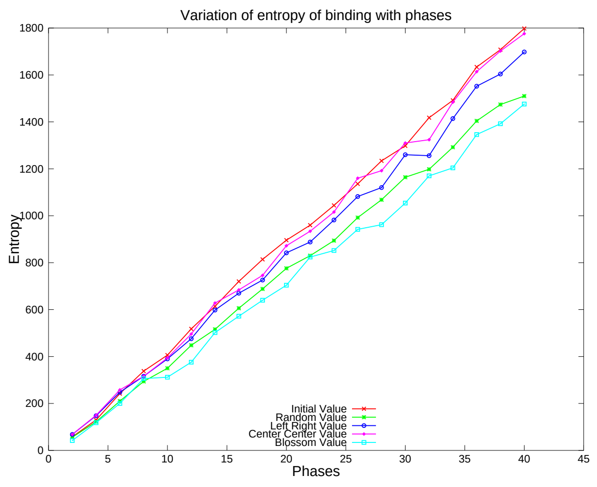


Fig. 11. **Variation of entropy with Phases**

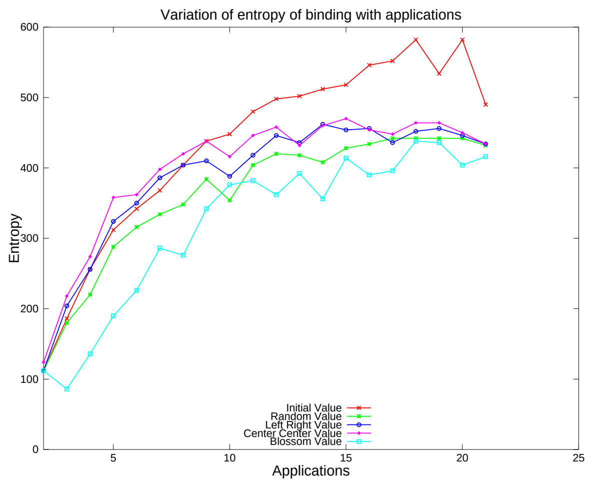


Fig. 12. **Variation of entropy with Applications**

[2] C.-B. Cho and T. Li, “Complexity-Based Program Phase Analysis and Classification,” in *Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques*. ACM, 2006, pp. 105–113.

[3] T. Sherwood, S. Sair, and B. Calder, “Phase Tracking and Prediction,” *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2, pp. 336–349, 2003.

[4] S. Banerjee, G. Surendra, and S. K. Nandy, “On the Effectiveness of Phase Based Regression Models to Trade Power and Performance using Dynamic Processor Adaptation,” *J. Syst. Archit.*, vol. 54, no. 8, pp. 797–815, Aug. 2008.

[5] C. yen Chang and P. Mohapatra, “Performance Improvement of Allocation Schemes for Mesh-Connected Computers,” *J. Parallel Distrib. Comput.*, vol. 52, no. 1, pp. 40–68, 1998.

[6] L. Wang and T. Jiang, “On the Complexity of Multiple Sequence Alignment,” *Journal of Computational Biology*, vol. 1, no. 4, pp. 337–348, 1994.