Міністерство освіти і науки України

Національний технічний університет України

«Київський політехнічний інститут ім. Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

ЛАБОРАТОРНА РОБОТА № 6-7

Виконав:

студент 5 курсу ФІОТ

групи ІО-з91МП

Гріщенко К. C.

Перевірила:

Русанова О. В.

Київ – 2020

**Мета роботи** – реалізувати другий етап планування шляхом призначення обчислювальних робіт по процесорах заданої КС.

**Вхідні дані –** граф задачі та побудована для нього черга обчислювальних робіт; граф КС (а для SMP - кількість процесорів та число банків оперативної пам’яті); два алгоритми призначення; параметри процесорів КС, які включають:

* число фізичних лінків;
* наявність процесорів вводу-виводу;
* дуплексність або напівдуплексність зв’язків;
* спосіб пересилки даних: пересилка повідомлень; конвейеризація пересилок пакетами (в цьому випадку необхідно задавати довжину пакету).

Алгоритми призначення обчислювальних робіт на процесори вибираються за погодженням з викладачем.

**Виконання**

За варіантом обрано алгоритм

Алгоритм призначення з урахуванням пріоритетів процесорів (по зв'язності). Даний алгоритм може використовуватись як для всіх обчислювальних робіт графа задачі, так і для призначення обчислювальних робіт першого ярусу графа задачі у поєднанні з алгоритмами «сусіднього» призначення

**Приклад сортування процессорів**

Приклад роботи алгоритму для системи з заданною топологією



Демонстрація результату сортування процесорів.

Изображение выглядит как снимок экрана

Автоматически созданное описание

Призначення виконання задачі на обчислювальну систему

Для прикладу виберемо задану топологію системи.

3

3

2

2



3

3

Граф задачі

1

3

1

101



**Изображение выглядит как компьютер

Автоматически созданное описание**

**Вихідний код**

if (allocationTask == null)

{

return BadRequest();

}

var nodesCount = allocationTask.Weights.Length;

var nodeNumber = 1;

var nodes = new *List*<*Job*>();

var tasksGraph = new *TaskGraph*();

foreach(var nodeWeight in allocationTask.Weights)

{

var node = new *Job*(nodeNumber, nodeWeight);

nodes.Add(node);

tasksGraph.AddTask(node);

nodeNumber++;

}

for(int i = 0; i < nodesCount; i ++)

{

for(int j = 0; j < nodesCount; j ++)

{

if(allocationTask.JobRelations[i][j] == 0)

{

continue;

}

var parentJob = nodes[i];

var childJob = nodes[j];

var weight = allocationTask.JobRelations[i][j];

childJob.AddParent(parentJob, weight);

parentJob.AddChild(childJob, weight);

}

}

var jobsSorter = new *CriticalPathByTime*();

var jobsOrder = jobsSorter.GetJobOrder(tasksGraph);

var allocationResult = new *AllocationResult*()

{

JobSorting = jobsOrder.Select(jobsOrder=> jobsOrder.Id).ToArray()

};

var processorCount = allocationTask.ProcessorRelations.Length;

var processors = new *List*<*Processor*>();

var computingSystem = new *ComputingSystem*();

for(var i = 0;i < processorCount; i ++)

{

var processor = new *Processor*( i + 1);

processors.Add(processor);

computingSystem.AddProcessor(processor);

}

for(int i = 0; i < processorCount; i++)

{

for(var j = 0; j < processorCount; j++)

{

var weight = allocationTask.ProcessorRelations[i][j];

if(weight == 0)

{

continue;

}

var parentProc = processors[i];

var childProc = processors[j];

parentProc.AddChild(childProc, weight);

childProc.AddParent(parentProc, weight);

}

}

var sorter = new *ProcessorSorter*();

var processorOrder = sorter.SortProcessors(computingSystem);

allocationResult.ProcessorSorting = processorOrder.Select(p=>p.Id).ToArray();

var tactNumber = 1;

var allocatedJobs = new *List*<*Allocation*>();

var ticksDesc = createInitialTickState(tasksGraph, computingSystem);

Debug.Write(string.Format("Tick {0}", tactNumber));

while(allocatedJobs.Count < tasksGraph.Count())

{

Debug.Write(string.Format("Tick {0}", tactNumber));

if(!ticksDesc.TryGetValue(tactNumber, *out* *TickState* tickState))

{

System.Console.Write("Tick info not found");

continue;

}

while(tickState.ReadyJobs.Any() && tickState.ReadyProcessors.Any())

{

var allocation = processTick(ticksDesc, tactNumber, processorOrder, jobsOrder.ToList(), allocatedJobs);

}

tactNumber ++;

addTickState(ticksDesc, tactNumber, new *List*<*Job*>(tickState.ReadyJobs), new *List*<*Processor*>(tickState.ReadyProcessors));

}

var schedule = new int [allocatedJobs.Max( j => j.Finish)][];

for(var i = 0; i < schedule.Length; i ++)

{

schedule[i] = new int[computingSystem.Count()];

}

foreach(var allocation in allocatedJobs)

{

for(var i = allocation.Start; i <= allocation.Finish; i++)

{

schedule[i - 1][allocation.Processor.Id - 1] = allocation.Job.Id;

}

}

allocationResult.Allocations = schedule.ToArray();

return allocationResult;

}

*private* *Dictionary*<int, *TickState*> createInitialTickState(*TaskGraph* tasksGraph, *ComputingSystem* computingSystem)

{

var readyJobs = new *List*<*Job*>();

var readyProcessors = new *List*<*Processor*>();

foreach(var job in tasksGraph)

{

if(!job.Parents.Any())

{

readyJobs.Add(job);

}

}

foreach(var proc in computingSystem)

{

readyProcessors.Add(proc);

}

var ticksDesc = new *Dictionary*<int, *TickState*>();

ticksDesc[1] = new *TickState*

{

ReadyJobs = readyJobs,

ReadyProcessors = readyProcessors

};

return ticksDesc;

}

*private* void addTickState(

*Dictionary*<int, *TickState*> ticksDesc,

int tickNumber,

*List*<*Job*> readyJobs,

*List*<*Processor*> readyProcessors)

{

if(!ticksDesc.TryGetValue(tickNumber, *out* *TickState* newTicState))

{

ticksDesc[tickNumber] = new *TickState*

{

ReadyJobs = new *List*<*Job*>(readyJobs),

ReadyProcessors = new *List*<*Processor*>(readyProcessors)

};

}

else

{

newTicState.ReadyJobs.AddRange(readyJobs);

newTicState.ReadyProcessors.AddRange(readyProcessors);

}

}

*private* *Allocation* processTick(*Dictionary*<int, *TickState*> ticksDesc,

int tickNumber,

*List*<*Processor*> processorOrder,

*List*<*Job*> jobsOrder,

*List*<*Allocation*> allocatedJobs)

{

if(!ticksDesc.TryGetValue(tickNumber, *out* *TickState* tickState))

{

System.Console.Write("Tick info not found");

return null;

}

var readyProcessors = tickState.ReadyProcessors;

var readyJobs = tickState.ReadyJobs;

var allocation = allocate(readyProcessors, processorOrder, readyJobs, jobsOrder.ToList(), tickNumber);

allocatedJobs.Add(allocation);

readyJobs.Remove(allocation.Job);

readyProcessors.Remove(allocation.Processor);

allocation.Job.Allocated = true;

var readyChildren = getReadyChildren(allocation.Job, allocatedJobs);

foreach(var readyChild in readyChildren)

{

var parents = readyChild.Parents.Select(p=>p.RelatedJob);

var parentAllocations = allocatedJobs.Where( a=> parents.Contains(a.Job));

var readyTick= parentAllocations.Max(p=>p.Finish);

addTickState(ticksDesc,

readyTick + 1,

new *List*<*Job*>{readyChild},

new *List*<*Processor*>{allocation.Processor});

}

return allocation;

}

*private* *List*<*Job*> getReadyChildren(*Job* job, *List*<*Allocation*> allocatedJobs)

{

var readyChildren = new *List*<*Job*>();

foreach(var child in job.Children.Select( r=> r.RelatedJob))

{

if(child.Parents.All(r => r.RelatedJob.Allocated) && !allocatedJobs.Select( a=>a.Job).Contains(child))

{

readyChildren.Add(child);

}

}

return readyChildren;

}

*private* *Job* getBestJob(*List*<*Job*> readyJobs, *List*<*Job*> jobOrder)

{

var besJob = jobOrder.FirstOrDefault( j => readyJobs.Contains(j));

return besJob;

}

*private* *Processor* getBestProcessor(*List*<*Processor*> processorOrder, *List*<*Processor*> readyProcessors)

{

var bestProcessor = processorOrder.FirstOrDefault( p=> readyProcessors.Contains(p));

return bestProcessor;

}

*private* *Allocation* allocate(

*List*<*Processor*> readyProcessors,

*List*<*Processor*> processorOrder,

*List*<*Job*> readyJobs,

*List*<*Job*> jobsOrder,

int tactNumber)

{

var bestJob = getBestJob(readyJobs, jobsOrder);

var bestProcessor = getBestProcessor(readyProcessors, processorOrder);

var allocation = new *Allocation*

{

Processor = bestProcessor,

Job = bestJob,

Start = tactNumber,

Finish = tactNumber + bestJob.Weight - 1

};

return allocation;

}

}