한국 퍼지 및 지능시스템학회 논문지 2000, Vol. 10, No. 5 (2000)409~416

다중프로그래밍 공유메모리 다중프로세서 시스템을 위한 퍼지 기반 프로세서 할당 기법

Fuzzy-based Processor Allocation Strategy for Multiprogrammed Shared-Memory Multiprocessors

김진일 · 이상구

Jin-Il Kim and Sang-Gu Lee

한남대학교 정보통신 멀티미디어공학부

요 약

공유메모리 다중프로세서 시스템은 전체적인 시스템 이용률을 높이기 위하여 병렬 작업시 시분할(time-sharing), 공간분할(space-sharing), 갱스케줄링과 같은 프로세서 자원 공유 기법을 사용한다. 최근에는 주어진 작업의 병렬 코드 부분의 실행을 위해서 시스템 작업부하를 기준으로 프로세서의 수를 동적으로 조절하는 루프단계 프로세스 제어(LLPC)할당 기법이 제안되었다. 이 기법은 작업에 가능한 많은 프로세서를 할당하기 때문에, 나중에 도착하는 작업의 병렬부분을 수행해야 할 프로세서를 남겨두지 않는다. 이러한 문제를 해결하기 위해, 본 논문에서는 작업부하량, 작업수행예상시간, 프로세스의 수을 퍼지화하여 시스템의 부하량에 따른 퍼지규칙으로 새로운 프로세서 할당 기법인 FPA(Fuzzy-based Processor Allocation)를 제안한다. 또한, 시스템의 과부하 없이 각 작업에 대한 최대한의 병렬 가능성을 제공함으로써 기존의 할당 기법에 비해 우수한 성능을 보인다.

ABSTRACT

In the shared-memory mutiprocessor systems, shared processing techniques such as time-sharing, space-sharing, and gang-scheduling are used to improve the overall system utilization for the parallel operations. Recently, LLPC(Loop-Level Process Control) allocation technique was proposed. It dynamically adjusts the needed number of processors for the execution of the parallel code portions based on the current system load in the given job. This method allocates as many available processors as possible, and does not save any processors for the parallel sections of other later-arriving applications. To solve this problem, in this paper, we propose a new processor allocation technique called FPA(Fuzzy Processor Allocation) that dynamically adjusts the number of processors by fuzzifying the amounts ofneeded number of processors, loads, and estimated execution times of job. The proposed method provides the maximum possibility of the parallism of each job without system overload. We compare the performances of our approaches with the conventional results. The experiments show that the proposed method provides a better performance.

1. 서 론

공유메모리 다중프로세서 시스템은 시스템 구조나 프로그래밍 모델 모두 전통적인 단일 프로세서 시스템과 유사하기 때문에, 고성능 범용 컴퓨터 서버에 많이 사용된다. 여러 작업이 동시에 실행되도록 프로세서를 공유하기 위해서, 전통적인 유닉스 시분할 기법을 확장하여 사용한다[3]. 그러나, 이 기법은 단일 프로세서 시스템에서는 효율성이 증명되었지만, 다중프로세서 시스템에서는 아직 원활히 적용되지 않는다. 이 기법에 대한 여러 가지 포괄적인 연구가진행되었지만, 높은 문맥전환(context switching) 오버헤드, 빈약한 캐시 이용률, 비효율적인 잠금(locking)과 동기화[6,9,12,15,18]와 같은 문제가 제기되었다. 또한, 공간분할(space-sharing 또는 space-partitioning), 타스크와 관계된 갱스케줄링(gang-scheduling)과 같은 프로세서들이 nonblocking 동기화[6]를 사용하여 프로세스[9,15]들을 동적으로 제어하고, 잠금(lock)을 포함하는 타스크가 교환되지 않도록 하는 정책들을 포함하도록 제안되었다. 그렇지만, 이러한 해결책은 전체적인 시스템 이용률은 향상되지만, 개별적인 작업에 대한 성능을 저하시키거나, 특별한 프로그래밍 모델을 가지는 운영체제에서는 수정을 필요로 한다.

최근에 제안된 루프단계 프로세스 제어(Loop-level process control, LLPC)[16] 할당방법은 작업에 생성되는 프로세스의 수를 제어함으로써, 작업이 시스템 과부하 없이 가능한 많은 프로세서들을 이용할 수 있도록 한다. 그러나, 이 기법에서는 가능한 많은 가용

프로세서를 할당하기 때문에, 나중에 도착하는 작업의 병렬 부분을 수행할 어떤 프로세서도 남겨두지 않는다. 비록 다른 작업이 프로세서의 할당받게 될지라도, LLPC 접근은 다른 작업들을 고려하지 않고 첫 번째 작업에 너무 치중하여 수행하는 단점을 가진다. 프로세서 할당 기법의 궁극적인 목적은 독립적인 작업들 사이의 간섭을 배제함으로써 각 작업 프로그램의 전체 실행 시간을 최소화하는데 있다.

본 논문에서는 LLPC 기법을 보완하기 위하여 퍼지 개념을 적용한 퍼지 기반 프로세서 할당 기법(Fuzzy-based Processor Allocation strategy, FPA)을 제안하고 비교, 분석함으로써 기존의 프로세서 할당 기법들[16,17]을 확장한다.

2. 관련연구

2.1 시스템 구조와 프로그래밍 모델

공유메모리 다중프로세서 시스템의 메모리는 모든 프로세서들이 동일하게 접근하기 때문에, 이런 시스템들은 전통적인 단일 프로세서 시스템과 유사한 프로그래밍 모델을 가진다. 따라서, 공유메모리 다중프로세서는 널리 상용화되었고, 고성능 범용머신에 사용된다. 이 시스템에서 모든 프로세서들은 같은 속도로 실행하고 각 프로세서는 다른 프로세서들과는 독립적으로 명령을 수행한다. 어떤 공유메모리 다중프로세서 시스템은 평균 메모리 접근 시간을 향상시키기 위해서 프로세서 모듈에 지역 캐시 메모리를 가진다. 공유메모리 다중프로세서 시스템을 위한 가장 일반적인 프로그래밍 모델중의 하나는 루프 단계(loop-level) 병렬화 모델이다.

프로그래머나 컴파일러는 작업의 독립적인 부분을 인식하고 병렬 타스크들로 나눔으로써 작업을 병렬화한다. 이러한 병렬 타스크들은 병행적으로 수행될 수 있다[8]. 이런 프로그래밍 모델로, 기존의 작업 프로그램을 병렬화하면 순서적인 코드를 재구성하지 않아도 된다. 이런 접근 방법이 작업 고유의 병렬성을 최대한으로 이용할 수는 없지만, 코드의 이식성을 증가시킬 수 있고 새로운 병렬 알고리즘을 개발하는데 있어서 프로그래머의 부담을 줄어들게 한다. 이러한 루프단계 병렬화 작업은 다음과 같은 순서로 수행된다. 작업 초기에 부 프로세스의 수는 일반적으로 프로세서의 수와 동일하게 생성된다. 그리고, 작업이 프로그램의 병렬부분에 도달했을 때, 각 부 프로세스는 병행적으로 병렬 작업의 각 할당 부분을 실행한다. 병렬 부분의 끝에서 모든 부 프로세스들은 동기화되고 주 프로세스(master process)는 다음 병렬부분에 도달할 때까지 순차적인 작업 부분을 계속해서 실행한다.

동기화 완료후 부 프로세스들을 소멸시키고 다음 병렬부분이 실행되기 시작할 때 새로운 프로세스들을 생성한다. 다른 방법은 부 프로세서들을 수면상태에 놓거나 다음 병렬 부분에서 필요로 할 때까지 공전상태로 둔다. 잠금(lock)과 풀림(unlock) 동작은 공유변수에 접근을 일치하도록 하거나 정확한 프로그램 실행을 유지하기 위해 사용된다.

2.2 기존의 프로세서 할당기법

작업의 특성에 따라 각 병렬 부분에서 요구하는 프로세스들의 수가 다르다. 효과적으로 프로세서를 공유하기 위해서, 프로세서에 타스크를 할당하는 스케줄링 기법이 필요하다. 가장 일반적으로 사용되는 방법은 전통적인 단일 작업 큐 시분할 접근으로 단일 프로세서 시스템인 유닉스 운영체제에서 사용되었다. 이 시분할 기법을 이용하면 여러 작업을 프로세서에 멀티플렉싱함으로써 시스템 이용률을 증가시킬 수 있지만, 개별적인 작업 성능을 심각하게 감소시킬 수 있다. 최근 연구에 의하면, 작업 성능은 문맥전환 오버헤드, 데이터 캐쉬 성능손실(data cache corruption), 과도한 잠금(locking) 또는 동기화 오버헤드에 의해 영향을 받는다[15].

프로세스 스래싱(process thrashing), 비효율적인 잠금 그리고 동기화 문제를 해결하기위한 다른 접근 방법은 관련된 모든 타스크들이 같은 시간에 프로세서 할당이 이루어지도록 하기 위해서, 작업에 관련된 타스크를 공동으로 스케줄링(coschedule)하는 것이다[12]. 비록 코스케줄링(coscheduling) 또는 갱스케줄링 기법이 프로세스 스래싱(process thrashing)를 피하고 동기화[5]을 사용하는 프로세스에 대해 성능 향상을 가져왔을지라도, 어떤 작업이 시스템의 모든 프로세서를 요구하지 않아 다른 작업 프로세스를 위해서 충분한 프로세서를 남겨두지 않을 때[6], 프로세서 분열은 여전히 발생한다. 이런 프로세서 분열은 컴퓨팅 자원의 25퍼센트 가량 손실을 발생시킨다. 더욱이, 문맥전환이 여전히 요구되고, 코스케줄링 기법은 여전히 캐시 성능 저하의 원인이 된다.[6].

문맥전환율을 줄이기 위한 가장 일반적인 방법은 프로세서들을 여러 개의 독립적인 영역들로 나눈 다음, 각 작업을 해당 영역에서 실행하는 것이다. 이런 형태의 공간분할(space-sharing) 또는 고정분할(static partitioning)은 프로세서가 작업들 사이에서 경쟁하는 요인을 제거함으로 대규모 병렬 시스템에 사용된다. 분할은 머신이나 운영체제에 따라 소프트웨어 또는 하드웨어로 행해질 수 있다. 프로세서들이 time-

multiplex을 필요로 하지 않기 때문에, 캐쉬 데이터는 유지될 수 있다. 이러한 접근은 과도한 잠금과 동기화 오버헤드를 방지한다. 그렇지만, 작업에 필요한 프로세서의 수가 실행 과정에서 다양하게 변화하기 때문에 평균 시스템 이용률은 낮아질 수 있다. 부가적으로, 작업이 생성할 수 있는 프로세스의 수를 제한하는 것은 다른 영역에 사용 가능한 유휴 프로세서(idle processor)을 이용할 수 없으므로 병렬성이 제한된다.

문맥전환 오버헤드를 최소화하는 동시에 시스템 이용률을 향상시키기 위해서, 시스템이 동적으로 분할할 수 있도록 하고 작업들 사이에서 공유하도록 한다[9, 15]. 이 기법으로, 만약 어떤 작업이 가능한 병렬화보다 적은 분할을 가지는 경우 외에는, 시스템 전체 프로세서를 각 작업에 대해 균등하게 할당한다. 비록 프로세서 할당을 동적으로 조정하여 시스템 이용률을 향상시키고 문맥전환율을 감소시킬 수는 있지만, 이러한 소프트웨어 해결책은 스레드(thread)의 손실로부터 복구할 수 있는 특별한 프로그래밍 모델과 운영체제의 수정을 필요로 한다. 더욱이, 어떤 실행 프로세스들은 프로세서 재 할당을 위해서 중지되기 때문에, 어떤 critical 프로세스들이 중지되는 경우, 다른 종속적인 프로세스에 영향을 미친다. 결과적으로, 동적인 중지와 재 할당 프로세서들은 시스템 오버헤드를 증가시키고 작업의 성능을 감소시킨다.

공유 메모리 다중프로세서 시스템에서 가장 일반적인 프로그래밍 모델 중의 하나가 루프단계 병렬화 모델이다. 이 프로그래밍 모델을 가지고, 많은 작업 프로그램의 병렬화가 순서 코드의 재배열 없이 개발될 수 있다. *LLPC* 할당방법[16]은 시스템 부하와 작업의 병렬성을 기반으로, 작업에 생성되는 프로세스의 수를 제어함으로써, 작업이 시스템 과부하 없이 가능한 많은 프로세서들을 이용할 수 있도록 한다. 한 작업이 병렬 부분에 도착했을 때, 현재 쓰이고 있는 프로세서의 수를 결정하기 위하여 작업부하(system_load)를 체크한다. 이 값을 사용하여, 시스템에서 전체 프로세서의 수(total_physical_p), 최대 병렬화를 얻기 위한 병렬부분의 프로세서의 수(no_needed), 그 작업에 할당될 수 있는 프로세서의 수(allow_p)는 다음과 같이 계산될 수 있다.

$$allow\_p = \max[1,\ \min(no\_needed,\ total\_physical\_p - system\_load + 1)];$$

그런 다음, 그 작업은 이 병렬부분을 실행하기 위해 allow_p 프로세스를 생성한다. 모든 병렬작업이 완료되었을 때, 그 작업은 하나의 프로세서를 사용하여 작업의 다음 순차적인 부분의 실행을 계속한다. 작업의 다음 병렬 부분에 도달하면, 위의 절차를 반복하여 수행한다.

위에서 기술된 LLPC 기법에서는 가능한 많은 가용 프로세서를 할당하기 때문에, 나중에 도착하는 작업의 병렬 부분을 수행할 어떤 프로세서도 남겨두지 않는다. 비록 다른 작업이 프로세서의 공유를 얻게 될 지라도, LLPC 접근은 다른 작업들을 고려하지 않고 첫 번째 작업에 너무 치중하여 수행하는 단점을 가진다.

3. 퍼지 기반 프로세서 할당 기법

시스템에서 실행 가능한 프로세스의 수를 동적으로 조절한다면 문맥 전환율을 줄일 수 있다. 이것은 캐시 데이터를 유지하는데도 도움을 주고 과도한 잠금 오버헤드를 방지한다. 루프단계(Loop-level) 병렬화 프로그래밍 모델에 프로세서의 수를 동적으로 제어하는데 퍼지 개념을 적용한 것이 본 논문에서 제안하는 퍼지 기반 프로세서 할당기법(FPA)이다. 이 기법의 기본 아이디어는 시스템 부하와 프로세서의 작업 실행 예상 시간을 기반으로, 작업에 생성되는 프로세스의 수를 퍼지 개념을 이용하여 제어함으로써, 작업이 시스템 과부하 없이 가능한 많은 프로세서들을 이용할 수 있도록 한다. 시스템 부하가 높을 때, 생성 가능한 최대의 프로세스 대신에 적절한 수의 프로세스를 생성하도록 함으로써 문맥전환율을 줄인다. 결과적으로, 하나의 작업이 모든 프로세서들을 독점할 수 없는 반면에, 실행은 모든 작업들에 대해 진행될 수 있다. 그리고, 시스템 부하가 낮을 때, 병렬 작업이 모든 유휴 프로세서를 이용하도록 한다.

3.1 퍼지화 시스템의 구성

그림 1은 퍼지 개념을 사용한 프로세서 할당기의 구성을 나타낸다. 어떤 작업의 병렬부분에 대한 작업 할당 요구가 입력으로 들어오면, 현재 수행중인 프로세서들의 수행 상태 정보들과 지식베이스에 나타나 있는 퍼지 규칙들을 사용하여 퍼지 추론을 하게된다. 추

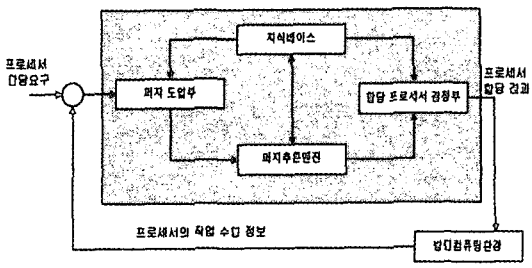

그림 1. 퍼지 프로세서 할당기의 시스템 구성

론결과에 의해 작업할당이 이루어질 프로세서의 수를 결정하게 된다.

먼저 퍼지 프로세서 할당기에서 사용하는 변수들과 이들의 퍼지화 방식에 대하여 살펴보자. 퍼지 프로세서 할당기에서 사용하는 변수들은 시스템의 작업부하량, 프로세서의 작업 수행 예상 시간, 필요한 프로세스의 수를 사용한다. 시스템의 작업 부하량은 시스템에서 활성화된 사용자 프로세스들의 전체의 수로 나타낸다. 즉, 활성화된 프로세서의 수의 많고 적음의 정도를 퍼지화하여 표현한다. 그리고 각 프로세서들의 작업 수행 예상 시간은 수행시간이 길고 짧음의 정도에 따라 퍼지화하여 표현된다.

3.2 프로세서 할당을 위한 퍼지 규칙

본 논문에서는 퍼지 개념을 이용하여 프로세서의 수를 제어하기 위한 퍼지 제어 시스템으로 제어 규칙을 위해 사용되는 입력 변수는 시스템의 작업부하량(system load : SL), 프로세서의 작업 수행 예상 시간(processor service time : PST), 출력변수로는 프로세스의 수(FP)로 하였다. 즉, 퍼지규칙은 전건부의 변수 2개, 후건부의 변수 1개이다.

각각의 입력변수와 출력변수의 퍼지 소속함수의 linguistic term은 각각 5개의 삼각형 퍼지 수로 분할된 퍼지공간상에 표현하였다. 시스템의 작업부하량(system load : SL)은 그림 2, 프로세스의 수(FP)는 그림 3과 같다. 각 퍼지소속함수는 [0, 1]사이의 값으로 정규화하여 표현하였다.

이와 같이 정의된 입력변수와 출력변수의 퍼지 집합을 이용하여 제안된 퍼지 프로세서 할당기를 위한

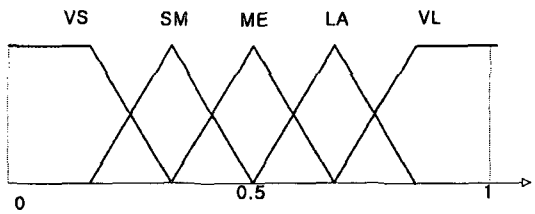

그림 2. 입력 변수에 대한 퍼지소속함수-시스템 부하

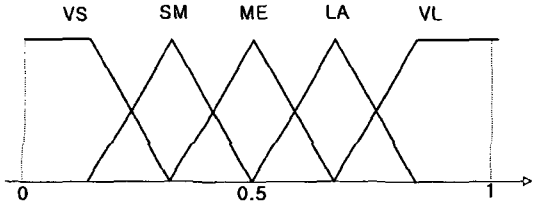

그림 3. 출력 변수에 대한 퍼지소속함수-프로세스의 수

표 1. 퍼지 생성 규칙

작업부하량 / 작업수행시간 (PST \ SL)	VS	SM	ME	LA	VL
VS	VL	LA	ME	SM	LA
SM	VL	LA	ME	SM	LA
ME	VL	LA	ME	SM	VS
LA	LA	ME	SM	VS	VS
VL	LA	ME	SM	VS	VS

퍼지 생성 규칙은 표 1과 같다.

제안된 퍼지 생성규칙에 사용된 전문가적인 지식은 현재 시스템의 작업부하량이 많은 경우에는 프로세서의 작업 수행 예상시간이 길면, 가능한 적은 수의 프로세서를 할당하고, 예상시간이 짧으면 가능한 많은 프로세서를 할당한다. 그리고, 시스템의 작업부하량이 적은 경우는 프로세서의 작업 수행 예상시간이 길면 많은 프로세서를 할당하고, 예상시간이 짧으면 더 많은 프로세서를 할당한다. 물론, 할당된 프로세서는 병렬부분을 수행하기 위해 생성되는 프로세스의 수를 넘지 않는다. 정의한 규칙들을 추론하기 위하여 Mamdani의 Min 연산을 사용하고, 결과에 대한 비퍼지화 방법으로는 무게 중심법을 사용한다.

3.3 퍼지 기반 프로세서 할당 알고리즘

퍼지 기반의 프로세서 할당 알고리즘 다음과 같은 단계로 수행된다.

단계 1. 병렬 부분을 실행하기에 앞서, 시스템 부하량을 체크한다. 시스템의 작업부하량은 시스템에서 활성화된 사용자 프로세스들의 전체의 수(allow_p)로 정의한다.

단계 2. 시스템 부하를 기준으로, 아래와 같이 병렬부분을 수행하기 위한 프로세서의 수를 결정한다. 변수 total_physical_p는 시스템에서 물리적인 프로세서의 전체 수이고, busy_P는 그 순간에 시스템의 부하이다. 그리고 FP는 퍼지 프로세서 할당기의 출력값 즉, 프로세스의 수이고, needed_p는 병렬 부분이 최대로 병렬로 수행되기위해 요구되는 프로세스의 수이다. 마지막으로, allow_p는 임의의 작업에 병렬 부분을 수행하기위해 생성되는 프로세스의 수이다.

$$allow\_p = \max[1, \min(needed\_P, FP, total\_physical\_p - busy\_P)]$$

단계 3. allow_p 프로세스들을 생성하고, 시스템 작업 큐에 추가한다.

단계 4. 프로세서가 프로세스에 할당되었을 때, 프

로세스들은 병렬 작업의 각 할당부분을 실행한다.

단계 5. 모든 병렬 타스크가 완료되었을 때, 프로세스들은 동기화되고 프로세서들은 시스템에 되돌려준다.

단계 6. 단일 프로세서를 사용하여 작업의 다음 순차적인 부분의 실행을 계속한다.

단계 7. 각 병렬 부분에 대해 단계 1에서 단계 6까지 반복한다.

다른 동적인 프로세스 제어와는 달리, 퍼지 프로세서 할당 알고리즘은 활성화된 프로세서들을 중지하지 않고도, 시스템의 부하량의 변화에 즉각적으로 적응할 수 있다. 단순하게 알고리즘이 구현되었기 때문에, 운영체제의 스케줄러에 어떤 변화도 요구하지 않으므로, 사용중인 시스템에 새로운 스케줄링 기법으로 대체할 수 있다.

4. 성능평가

4.1 실험 환경

본 실험은 Tru64 UNIX 4.0F 운영체제로 동작하는 HPC160/320 시스템에서 실행한다. 이 시스템은 크로스바 스위치 공유 메모리 구조를 가지고 있고 HPC160은 16개, HPC320은 32개의 CPU로 구성되어 있다. HPC160은 500 MHz클럭로 동작되는 64-bit Alpha 21264 프로세서 4개를 연결한 AlphaServer ES40 building block를 기반으로 한다. 각 PE는 원칩 데이타 캐시 64 KB와 원칩 명령캐시 64 KB을 가진다. 2차 캐시는 4 MB 단일화된 명령과 데이타 캐시를 가진다. 시스템은 4개와 8개의 build block으로 구성되었을 때, 4-way interleaved 물리적 메모리, HPC160 시스템은 8 GB, HPC320은 16 GB 메모리를 가지며, 각각 64 GB, 128 GB까지 확장할 수 있다.

벤치마크 프로그램은 OpenMP Fortran 명령어를 사용하여 프로그램을 작성한 다음, 이를 KAI(Kuk & Associates Inc.) tool의 *Guide* 입력으로 사용하면, *Guide* 출력은 스레드와 Guide제공 라이브러리를 사용하여 구성된 병렬화된 Fortran 프로그램이 생성된다. 이 출력은 일반적인 Fortran 컴파일러를 사용하여 컴파일한다. 그런 다음, KAI 툴(tool)의 *GuideView*을 사용하여 성능을 분석한다. *GuideView*는 프로그램의 병렬 실행의 자세한 성능을 윈도우로 나타내는 그래픽 툴이다.

4.2 퍼지 기반 프로세서 할당 알고리즘 구현

FPA 알고리즘은 운영체제와 작업사이에서 구현되므로, 운영체제 또는 컴파일러의 수정이 필요없다. 이 알고리즘을 독립적인 1개의 PE에서 수행한다. 주어진 작업이 Guide를 통하여 병렬화된 후, FPA call은 그림에서 보는 바와 같이 주로 병렬 시작부분에 삽입된다. 그런 다음, Fortran 컴파일러로 실행파일을 생성하기 위해 컴파일된다.

그림 4에서 함수 FPA_numthreads( )은 현재 사용가능한 프로세서의 수를 결정한다. 그럼 다음, 프로세스의 수는 OMP_GET_NUM_THREADS( ) 서브루틴을 사용하여 결정한다. 이 시점에서, 새로 생성된 프로세스들은 운영체제에 의해 유지되는 시스템의 시분할 작업 큐로 보내진다. 병렬 루프 반복은 chuk 스케줄링을 사용하여 공평하게 이들 프로세스들에게 분산된다. 각 프로세서들은 「N/allow_p」 반복한다. 여기서, *N*은 병렬 루프에서의 전체 반복 수이다. 루프 실행이 완료된 후에 시스템 부하를 갱신하고 프로세스들을 반납한다. 현재 시스템 부하는 프로세서에 대해 실행중이거나 대기중인 사용자 프로세스들의 전체 수로 한다.

```
allow_p = FPA_numthreads()
!$OMP PARALLEL PRIVATE(allow_p)
CALL OMP_GET_NUM_THREADS(allow_p)
!$OMP DO 100 I = 0, 10000
.....
100 CONTINUE
!$OMP ENDDO

!$OMP END PARALLEL
```

그림 4. FPA 서브루틴의 위치

```
Do I = 1, 100
 ! serial section
 c$omp single
  Do J = 1, i_s
   D0 K = 1, size
          CALL INPUT(X)
          CALL INPUT(Y)
          CALL FLOAT_MUL(Z, X, Y)
          CALL OUTPUT(Z)
   ENDDO
  ENDDO
 ! parallel section
 c$omp parallel do
  Do J = 1, i_p
   D0 K = 1, size
          CALL INPUT(X)
          CALL INPUT(Y)
          CALL FLOAT_MUL(Z, X, Y)
          CALL OUTPUT(Z)
   ENDDO
  ENDDO
```

그림 5. 복합 벤치마크 프로그램

4.3 복합 벤치마크 성능평가

퍼지 프로세서 할당 알고리즘의 동작을 평가하기 위해 복합 벤치마크(synthetic benchmark)를 사용한다. 복합 벤치마크의 병렬적 특성은 그림 5에서 보는 것처럼, 시리얼과 병렬 파라메터를 모두 가지고 있다.

이 벤치마크 프로그램은 단일 프로세서로 수행되는 블록과 다중 프로세서로 수행되는 블록을 구성되어 있으며, 파라메타 i_s와 i_p를 통해서 정밀하게 병렬 특성들을 제어한다. 여기서, i_s는 순차 루프에서 반복문들의 수, i_p는 병렬 루프에서 반복문들의 수이다. 파라메타 i_s와 i_p는 각각 순차 작업 W_s와 병렬 작업 W_p의 양에 정비례한다. 예를 들면, 만약, i_p가 i_s보다 크다면, 그 때 벤치마크는 높은 병렬성을 가지므로 병렬로 수행될 때 높은 속도향상을 가져온다. 반대로, 만약 i_p가 i_s보다 작다면, 작업은 낮은 병렬성을 가지며 병렬로 수행될 때의 이점은 작다.

4.4 성능 비교

그림 6, 그림 7, 그림 8은 각각 4개, 8개, 16개의 프로세서에서 각 벤치마크 프로그램의 여러가지 병렬성에 대하여 실험한 결과이다.

4개의 프로세서를 사용하는 경우, 그림 6에서 보는

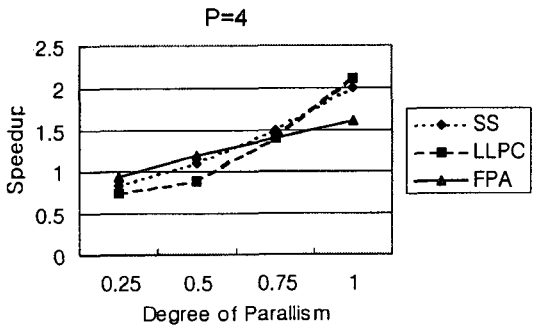

그림 6. P=4 프로세서 할당기법의 성능 비교

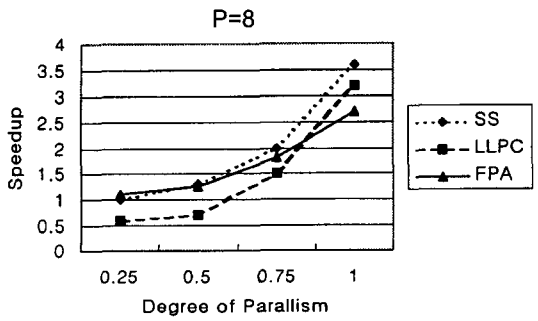

그림 7. P=8 프로세서 할당기법의 성능 비교

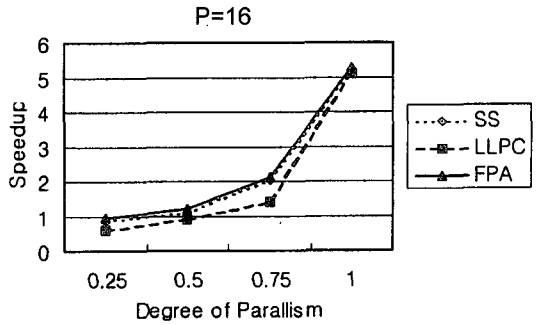

그림 8. p=16 프로세서 할당기법의 성능 비교

것 처럼, 병렬성의 정도가 높을 때, LLPC기법이 공간분할(space-sharing)기법보다 더 잘 수행되는 것을 보여준다. FPA기법은 낮은 병렬성을 가져도 작업을 잘 수행하는 것을 볼 수 있다. 더우기, 공간분할(space-sharing)기법 처럼 어떤 파티션내로 작업을 한정하지 않는다.

8개의 프로세서를 사용하는 경우, 4개의 프로세서을 가지는 개별적인 파티션에서 수행되는 각 작업을 수행하는 공간분할(space-sharing)기법은 병렬성 정도가 많은 경우 작업에 대해 최상의 성능을 보여준다. 병렬성 정도가 높을 때, LLPC 기법은 FPA기법보다 병렬 작업에 더 많은 프로세서를 할당한다. 그러나, 이 경우에, LLPC 기법이 순차적인 부분이 많을 경우, 공간분할(space-sharing)기법보다 더 낮은 성능을 보여준다.

16개의 프로세서를 사용하는 경우, 공간분할(space-sharing)기법은 작업의 병렬성 정도가 낮을 경우에도 좋은 성능을 나타낸다. 그렇지만, 작업 내에 수행할 병렬 작업이 많을 경우에는 FPA기법이 최상의 성능을 낸다. FPA기법을 적용하는 시스템이 대규모 시스템 일수록, 다른 작업을 위해 단지 소수의 프로세서를 남겨두고 가능한 많은 프로세서를 이용하기 때문에 전제적인 다중 프로래밍 작업의 성능이 높아진다. 또, 공간분할(space-sharing)기법이 잘 동작하도록 하기 위해서는 시스템이 어떻게 프로세서들을 고정적인 부분으로 나누어 작업을 실행하는지를 알아야한다. 만약 병행 작업의 수가 선지식으로 알려지지 않는 경우라면 제안된 FPA기법은 보다 더 확실한 해결 방법이고 작업부하의 변화에 따라 프로세서 할당을 동적으로 조절할 수 있다.

5. 결론 및 향후 연구

본 논문에서는 공유 메모리 다중프로세서 환경에서 퍼지 기반 프로세서 할당 알고리즘을 연구하였다.

LLPC기법은 어플리케이션 레벨에서 프로세서 파티션을 동적으로 조절하는 아이디어을 기반으로 하고 있다. LLPC 기법을 더 발전시킨 형태의 FPA기법을 제안하였다. 제안된 FPA기법은 두 개의 작업 프로그램을 동시에 수행했을 때, 공유메모리 HPC160/320 시스템상에서 단순한 LLPC기법과 비교하여 평가하였다. 작업의 병렬성 정도, 시스템의 크기, 프로세서 할당기법과 성능사이의 관계를 조사하였다. 그 결과, 공간분할(space-sharing) 기법과 LLPC 기법은 LLPC가 높은 병렬성을 가지는 작업에 대해 보다 큰 장점을 가지는 것을 제외하고는 대체적으로 비슷한 성능을 보였다. FPA기법은 프로세서를 동적으로 파티션할 뿐만 아니라, 나중에 도착하는 작업에 대해 적은 수를 프로세서를 남겨두기 때문에 가장 우수하게 동작한다.

향후 연구로는 다양한 변수들을 사용하거나 혹은 여러 환경에 적합한 작업 프로세서 할당기법에 대하여 연구가 필요하다. 또한, 분산 메모리 환경에서 퍼지 개념을 적용한 보다 유연한 프로세서 할당기법에 대한 연구가 필요하다.

참고문헌

[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy, "Scheduler Activations: Effective Kernel Support for User-Level Management of Parallelism," *ACM Trans. Computer Systems*, Vol. 10, No.1, pp. 53-79, Feb. 1992.

[2] Michel Banatre, Alain Gefflaut etc, "An Architecture for Tolerating Processor Failures in Shared-Memory Multiprocessors," *IEEE Trans. on Computer,* Vol. 45, No. 10, Nov. 1997.

[3] J. Barton and N. Bitar, "A Scalable Multi-Discipline, Multiple-Processor Scheduling Framework for IRIX," Proc. IPPS '95 workshop Job Scheduling Strategies for Parallel Processing, pp. 24-40, Apr. 1995.

[4] Convex Architecture Reference Manual(C-series). Convex Computer Corp., 1992.

[5] D. Feitelson and L. Rudolph, "Gang Scheduling Performance Benefits for Fine-Grain Synchronization," *J. Parallel and Distributed Computing*, Vol. 16, pp. 306-318, 1992.

[6] A. Gupta, A. Tucker, and S. Urushibara, "The Impact of Operating System Scheduling Policies and Synchronization Methods on the performance of Parallel Applications," *Proc. Conf. Measurement and Modeling of Computer Systems,* Vol. 19, pp. 120-132, 1991.

[7] C. J. Kim, "An Algorithmic Approach for Fuzzy Inference," *IEEE Trans. on Fuzzy Systems*, Vol. 5, No. 5, Nov. 1997.

[8] D. Lilja, "Exploiting the Parallelism Available in Loops," Computer, Vol. 27, No. 2, pp. 13-26, Feb. 1994.

[9] V. Naik, S. Setia, and M. Squillante, "Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments," Supercomputing, pp. 824-833, 1993.

[10] C. Natarajan, S. Sharma, and R. Iyer, "Impact of Loop Granularity and Self-Preemption on the Performance of Loop Parallel Applications on a Multiprogrammed Shared-Memory Multiprocessor," *Proc. 1994 Int. Conf. Parallel Processing,* Vol. II, pp. 174-178, Aug. 1994.

[11] Thu D. Nguyen, Raj Vaswani, and John Zahorijan, "Maximizing Speedup Through Self-Tuning of Processor Allocation," *Proc. of the 10th International Parallel Precessing Symposium*, pp. 463-468, April 1996.

[12] J. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proc. Distributed Computing Systems Conf.*, pp. 22-30, 1982.

[13] C. Severance, R. Enbody, "Comparing Gang-Scheduling with Dynamic Space Sharing on Symmetric Multiprocessors Using Automatic Self-Allocating Threads(ASAT)," *Proc. Int. Conf. Parallel Processing Symp.*, pp. 288-292, 1997.

[14] C. Severance, R. Enbody, S. Wallach, and B. Funkhouser, "Automatic Self-Allocating Threads on the Convex Exemplar," *Proc. Int. Conf. Parallel Processing,* Vol. I, pp. 24-31, 1995.

[15] A. Tucker, "Efficient Scheduling on Multiprogrammed Shared-memory Multiprocessors," Ph.D thesis, Dept. of Computer Science, Standford Univ., 1993.

[16] K. Yue and H. Lilja, "Loop-Level Process Control: An Effective Processor Allocation Policy for Multiprogrammed Multiprocessor Systems," Job Scheduling Strategies for Parallel Processing, D. Feitelson and L. Rudolph, eds., Lecture Notes in Computer Science, Vol. 949, pp. 182-199. Springer-Verlag, 1995.

[17] K. K. Yue and D. J. Lilja, "An Effective Processor Allocation Strategy for Multiprogrammed Shared-Memory Multiprocessors," *IEEE Trans.* Vol. 8, No. 12, Dec. 1997.

[18] J. Zahorian, E. Lazowska, and D. Eager, "Spinning Versus Blocking in Parallel Systems with Uncertainty," *Proc. int. Seminar Performance Distributed and Parallel Systems*, pp. 455-472, 1988.

김 진 일 (Jin-Il Kim)

1991년 : 한남대학교 컴퓨터공학과 학사
1993년 : 한남대학교 컴퓨터공학과 석사
2000년 : 한남대학교 컴퓨터공학과 박사
관심분야 : 컴퓨터구조, 퍼지 제어, 병렬 처리

이 상 구 (Sang-Gu Lee)

제9권 5호 참조